Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Dunja Rosina

Merjenje temperature v senzorskih omrežjih

diplomsko delo

univerzitetni študijski program računalništvo in informatika

Mentor: prof. dr. Nikolaj Zimic

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Hiter razvoj tehnologije je omogočil izdelavo zelo učinkovitih elementov senzorskih omrežij, zato so ta postala vedno bolj razširjena. Učinkovitost se odraža predvsem v majhni porabi energije in dolgi avtonomni življenjski dobi posameznega dela omrežja ter posledično celotnega omrežja.

V diplomski nalogi primerjajte dva operacijska sistema za senzorska omrežja in sicer Contiki in TinyOS. Na osnovi izbranega operacijskega sistema izdelajte prototip senzorskega omrežja, ki bo omogočal zajemanje temperature okolice. Senzorsko omrežje povežite z internetom. Poleg tega izdelajte še aplikacijo na mobilnem telefonu, ki bo omogočala prikazovanje izmerjene temperature.

Izjava o avtorstvu diplomskega dela

Spodaj podpisana Dunja Rosina, z vpisno številko **63020136**, sem avtorica diplomskega dela z naslovom:

Merjenje temperature v senzorskih omrežjih

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelala samostojno pod mentorstvom prof. dr. Nikolaja Zimica,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 1.10.2014 Podpis avtorice:

Olgi in Noniju

Na prvem mestu se iskreno zahvaljujem mentorju prof. dr. Nikolaju Zimicu za strokovno pomoč in usmerjanje pri izdelavi diplomskega dela. Posebna zahvala gre Martinu in Nuši, za pomoč pri pisanju diplomskega dela pa se zahvaljujem tudi Petri, Ani in Sandiju.

Kazalo

Povzetek

Cilj diplomskega dela je predstaviti brezžična senzorska omrežja, načrtovati in izdelati tako omrežje. Diplomsko delo je razdeljeno na teoretični in praktični del. V prvem delu opišemo brezžična senzorska omrežja in opredelimo probleme, ki jih predstavlja načrtovanje in izdelava teh omrežij. Nato podrobneje predstavimo element Zolertia Z1, strojno opremo, ki jo kasneje uporabimo v praktičnem delu. Sledi opis operacijskih sistemov in programiranja za brezžična senzorska omrežja, kjer izpostavimo dva najpogosteje uporabljena operacijska sistema za brezžična senzorska omrežja TinyOS in Contiki. Oba operacijska sistema preizkusimo in na kratko primerjamo. V praktičnem delu diplomskega dela izdelamo prototip brezžičnega senzorskega omrežja. Z omrežjem zajemamo temperaturo in podatke iz omrežja prikažemo s pomočjo aplikacije na pametnem telefonu.

Ključne besede:

brezžično senzorsko omrežje, BSO, Contiki, TinyOS, merjenje temperature, energijska učinkovitost

Abstract

The following thesis aims to present wireless sensor networks and to propose a design as well as construct a practical example of one such network. The thesis addresses the matter in two parts with the theoretical part being followed by the practical part. The theoretical part describes and defines wireless network sensors, along with problems occuring while designing and building these networks. Firstly, the Zolertia Z1 module and platform, used in the practical part, are presented in theory, followed by a description of operating systems and programming of wireless sensor networks with emphasis on two of the most commonly used operating systems TinyOS and Contiki, both of which are tested and briefly compared. In the practical part of the thesis a prototype of a wireless sensor network is set up. The prototype network gathers temperature data, which is then displayed by means of a smartphone application.

Keywords:

wireless sensor network, WSN, Contiki, TinyOS, temperature measurement, energy efficiency

Poglavje 1

Uvod

Hiter razvoj na področju informacijske tehnologije in brezžične komunikacije ter vse manjša, zmogljiva in poceni strojna oprema so pripomogli k širjenju in vse večji uporabi t.i. brezžičnih senzorskih omrežij. Tako omrežje je sestavljeno iz posameznih elementov, ki so sposobni interakcije z okoljem (s pomočjo zaznavanja in uravnavanja določenih parametrov okolja). Za tako omrežje je praviloma pomembna zelo nizka poraba energije, v splošnem pa se med seboj razlikujejo, saj jih lahko uporabimo za reševanje popolnoma različnih problemov. Če navedemo le nekaj različnih možnih uporab takega omrežja, ki jih navaja strokovna literatura: spremljanje premikov živali v gozdu, uravnavanje vlažnosti tal, spremljanje jakosti gozdnega požara, meritve na nedostopnih območjih, kjer lahko raztrosimo elemente z letala – na primer spremljanje aktivnosti podmorskih vulkanov, vojšaki nameni, ipdr.

Ravno zaradi izredno velikega števila možnosti uporabe takega omrežja, le-ta predstavljajo velik izziv za raziskovalce in inženirje, ki taka omrežja načrtujejo. Ustrezna predstavitev brezžičnih senzorskih omrežij zahteva znanje in pregled zelo velikega števila različnih področij računalništva, od strojne opreme, radijske komunikacije do operacijskega sistema. Zaradi izredno velike raznovrstnosti brezžičnih senzorskih omrežij ne poznamo niti ene same univerzalne tehnične rešitve. Elementi, priključeni v tako omrežje so lahko vsi enake vrste, ali pa se med seboj razlikujejo. Število elementov je lahko

majhno ali veliko. Lahko je za nas pomembnejša cena omrežja ali morda življenjska doba posameznega elementa. Vprašanje je tudi, ali potrebujemo izredno natančne podatke iz okolja ali ne, ali bomo elemente lahko čez čas zamenjali ali je to nemogoče.

Kljub vse večji popularnosti brezžičnih senzorskih omrežij je zelo težko najti literaturo, ki bi v splošnem povzela vse osnovne značilnosti brezžičnih senzorskih omrežij. Večina literature se po kratkem uvodu zelo hitro spusti v točno določeno tematiko.

V teoretičnem delu diplomskega dela bomo zato predstavili osnovne karakteristike in zahteve takih omrežij. Poglavje bo sestavljeno kot splošni pregled in prestavitev brezžičnih senzorskih omrežij. Ta del bo namenjen tudi bralcu, ki se s to tematiko šele spoznava.

Temu bo sledilo poglavje o strojni opremi. Tam bomo predstavili element Zolertia Z1, ki ga bomo v zadnjem, praktičnem delu diplomskega dela uporabili za izdelavo prototipnega brezžičnega senzorskega omrežja.

V četrtem poglavju bomo predstavili programsko opremo, ki je primerna za brezžična senzorska omrežja. Podrobneje bomo opisali operacijska sistema TinyOS in Contiki, ki sta najpogosteje uporabljena operacijska sistema za izdelavo brezžičnih senzorskih omrežij. Na kratko ju bomo tudi primerjali.

Praktični del diplomskega dela bo predstavljen v petem poglavju. V njem bomo opisali izdelavo prototipnega omrežja. Povezali bomo štiri elemente v brezžično senzorsko omrežje, ki ga bomo spremljali in nadzorovali s pomočjo aplikacije na pametnem telefonu.

Poglavje 2

Brezžična senzorska omrežja

V tem poglavju bomo predstavili osnovne značilnosti brezžičnega senzorskega omrežja (BSO v nadaljevanju), način povezovanja takega omrežja z okoljem, zahteve za energijsko učinkovitost, robustnost in uporabljeno strojno opremo.

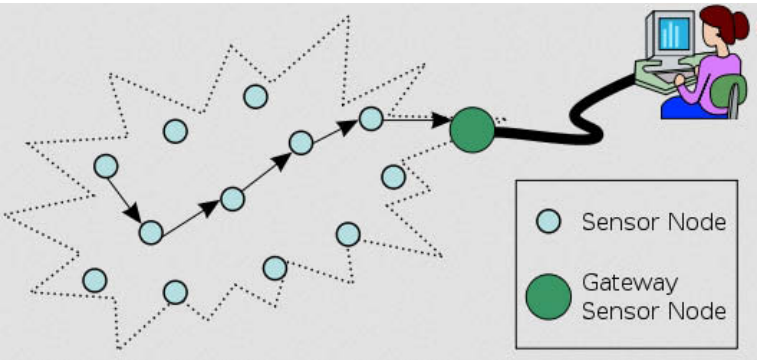
V splošnem lahko tako omrežje opišemo kot omrežje, sestavljeno iz elementov, ki med seboj komunicirajo. Vsa komunikacija v takem omrežju poteka brezžično. Elementi omrežja zaznavajo okolico in nanjo lahko vplivajo, njihovo število pa ni omejeno. Velikost omrežja je poljubna.

Lahko govorimo o omrežju z enim ali več tisoč elementi, odvisno od namena uporabe. Elementi zaznavajo dogodke v okolici ali spremembe okolja, zbirajo podatke iz okolice in jih pošiljajo do ponora (angl. sink). Poleg zaznavanja pa lahko elementi tudi vplivajo na okolje in posredno spremenijo določeno količino v okolju (npr. vlažnost zemlje s pomočjo škopilcev vode).

V pričujočem diplomskem delu bomo fizično senzorsko vozlišče (v angleški literaturi: mote[1]) BSO imenovali element omrežja. Element omrežja je sestavljen iz petih komponent: procesorja, pomnilnika, senzorjev, oddajnika in sprejemnika in modula za napajanje. To predstavlja glavno strojno opremo za realizacijo BSO. Podrobneje element omrežja predstavimo v podpoglavju 2.1.

Elementi BSO so s svojim okoljem povezani preko senzorjev. Senzor je

[1]Mote v slovenščino sicer dobesedno prevajamo kot droben delček, drobec



Slika 2.1: Shematični prikaz BSO, vir [5]

element, ki proizvede na izhodu signal, ki enolično ustreza vrednosti opazovane količine (temperatura, vlažnost, tlak) na vhodu senzorja. Ta signal interpretiramo (ga prevedemo v človeku razumljivo obliko) glede na navodila proizvajalca naprave. Na tak način lahko zaznavamo različne količine prisotne v okolju omrežja, od zvokov, nivojev gladine vode, magnetnega polja do temperature. Na količine v okolju lahko tudi vplivamo, na primer akvariju ustrezno prilijemo ali odlijemo vodo glede na izmerjen nivo gladine vode v njem.

Elementi omrežja morajo biti robustni, energijsko učinkoviti in poceni. Ker mora omrežje z elementi delovati tudi več let (pričakovana življenjska doba omrežja je odvisna od namena uporabe), sta robustnost in energijska učinkovitost ključnega pomena, saj se elementi zanašajo na omejen vir energije (baterije). V večini primerov je namreč nemogoče zagotoviti neprekinjen vir napajanja (npr. sledenje živalim v gozdu) ali menjave elementov omrežja. Kako pomembna je cena elementa se pokaže v primeru, da potrebujemo izredno veliko število elementov v enem omrežju (npr. več kot 1000).

Omenili smo, da elementi BSO zaznavajo ali vplivajo na dogodke (ali količine) v okolici. To je ključnega pomena za razumevanje BSO, saj nas v takem omrežju zanimajo dogodki in okolica. Želimo prejeti odgovor na določeno vprašanje v pravem trenutku. Vprašanju kaj si želimo s pomočjo BSO izvedeti, moramo prilagoditi komunikacijo med elementi in način, kako

pridobivamo podatke:

Zanima nas konkreten dogodek Ali je povprečna temperatura stadiona presegla določen prag? Ali je vlažnost tal na določenem območju gozda padla pod prag, obstaja nevarnost požara? Ko se zgodi določen dogodek, nas omrežje o tem obvesti.

Zanima nas spreminjanje količine skozi določeno časovno obdobje V danem časovnem intervalu konstantno opravljamo ustrezne meritve in zbiramo podatke. Ali se erozija tal povečuje na določenem območju? Kako se premikajo gozdne živali pozimi?

Prejeti želimo podatke iz omrežja le na zahtevo Kolikšna je temperatura v tem trenutku? Kje se trenutno nahaja trop divjih živali, ki jih spremljamo? Omrežje se odzove zahtevi uporabnika in na podlagi zahteve vrne željene podatke.

Prenos podatkov v takem omrežju je zelo pomemben, vendar je v ospredju odgovor na vpašanje, kaj želimo s pomočjo BSO izvedeti. Ker za prenos podatkov potrebujemo veliko energije, je ključnega pomena, da po omrežju poteka promet le takrat, ko je to nujno potrebno. Omrežja ne smemo poplaviti[2] ali preobremeniti, saj se s tem troši energija elementov in skrajšuje življenjska doba omrežja. Energijska učinkovitost celotnega BSO, ne le elementov omrežja, je tako zelo pomembna. Čeprav sam element omrežja porabi le malo energije v stanju mirovanja, se lahko zgodi, da zaradi pretiranega in nespametnega pošiljanja podatkov po omrežju poraba zelo naraste. Ta problem rešujemo z izbiro načina komunikacije, ki mora ustrezati namenu uporabe BSO.

BSO mora biti, tako kot njegovi elementi, v celoti robustno. To pomeni, da mora biti odporno na napake (npr. morebitno odpoved elementa) in do neke mere sposobno re-konfiguracije brez zunanjih vplivov. Ko enkrat

[2]Poplavljanje omrežja je pošiljanje paketov vsem sosednjim elementom, vsak element pošlje vsako sporočilo vsem sosednjim elementom, izvzet je le element, ki je poslal izvorno sporočilo

omrežje "postavimo"(npr. raztrosimo elemente po morskem dnu) zahtevamo, da deluje vsaj toliko časa, kolikor smo predvideli.

V zgoraj omenjenem primeru, da elemente omrežja razstrosimo po morskem dnu, težko zahtevamo enakomerno geografsko razporeditev ali točno določeno strukturo omrežja. Razporeditev elementov je v BSO popolnoma pogojena namenu uporabe.

2.1 Element brezžičnega senzorskega omrežja

Eden glavnih izzivov pri načrtovanju BSO je izdelava (oziroma izbira) ustreznega elementa, ki je energijsko zelo učinkovit. Pri izbiri telefona za mobilno (telefonsko) omrežje nimamo težav, čeprav so prav tako del omrežja, so prenosljivi, majhni in energijsko učinkoviti. Pri mobilnih telefonih se namreč smatra teden dni delovanja z baterijo brez vmesnega polnjenja za dolgo dobo, nas pa zanima življenjska doba izmerjena v mesecih oziroma letih. Tako je zelo pomembna energijska učinkovitost vseh komponent elementa. Komponente so prikazane na sliki 2.2.
Pet ključnih komponent, ki sestavljajo element:

procesor običajno je to MCU (mikrokontroler) lahko pa tudi FPGA (Field-Programmable Gate Array)

(dodaten) pomnilnik za shranjevanje programov in podatkov[3]

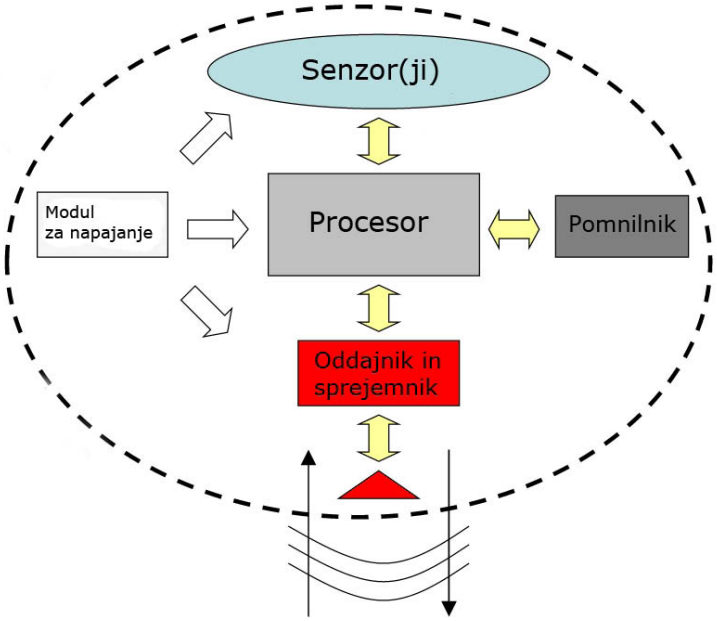
senzor(ji) s katerimi opazujemo oziroma tipamo okolje

oddajnik in sprejemnik potrebujemo za povezavo v omrežje in prenos podatkov

modul za napajanje

Procesor predstavlja jedro elementa. Zbira in procesira podatke senzorjev, nadzouje pošiljanje in prejemanje paketov. Izvaja programsko opremo, ki

[3]odvisno od izbranega mikrokontrolerja in namena uporabe BSO



Slika 2.2: Komponente elementa, vir [2]

jo za delovanje BSO potrebujemo. Rečemo lahko, da je ta komponenta centralno procesna enota elementa. Izbira ustreznega procesorja je pomembna, saj ne smemo zanemariti porabe energije nobene komponente. Tako uporaba splošno namenskega procesorja, ki se uporabljajo v osebnih računalnikih (npr. Intel i7) ali SoC (system on a chip) ki se uporablja v mobilnih telefonih (npr. Qualcomm Snapdragon) ni smiselna, saj je poraba prevelika, da bi lahko ustrezali zahtevam BSO. Povprečna poraba i7 procesorja je v obsegu okoli 200 W, Snapdragon-a, ki se uporablja predvsem v pametnih telefonih, pa v obsegu 2 W. Zato v elementih BSO največkrat srečamo mikrokontorlerje, ki so prilagojeni delu v vgrajenih sistemih (angl. embedded systems) in omogočajo enostavno povezovanje drugih naprav. Poraba energije mikrokontrolerjev je majhna, v obsegu 10^{-4} W. Ponavadi imamo na voljo več načinov delovanja, ki omogočajo uporabo le majhnega dela kontrolerja ali pa popolno mirovanje. Druga možnost, ki jo imamo pri izbiri procesorja na voljo je FPGA. FPGA lahko reprogramiramo in prilagajamo zahtevam, ki

jih mora izpolnjevati omrežje.

Brez senzorjev, ki so ena od ključnih komponent elementa bi bilo BSO povsem brez pomena. S pomočjo senzorjev zajemamo podatke iz okolice, in ti podatki pomenijo glavni smisel uporabe BSO. Izbira senzorjev je pogojena z namenom uporabe.
V grobem lahko senzorje razdelimo v tri večje skupine [1]:

pasivne, neusmerjene senzorje: ti lahko merijo količine, ne da bi vplivali na samo okolico, zato rečemo, da so pasivni. Smer meritve nima nobenega vpliva na dejansko izmerjeno količino (temperatura, vlažnost, detekcija dima).

pasivne, usmerjene senzorje: podobno kot prejšnja skupina so pasivni in na okolico ne vplivajo, vendar pa je smer meritve pomembna (zajem slike ali videa v določeni smeri)

aktivne senzorje: senzorji vplivajo na okolje z dejanjem meritve (aktivni sonar ustvarja zvočne valove in spremlja njihov odboj)

Oddajnik in sprejemnik potrebujemo za komunikacijo in izmenjavo podatkov med elementi. Najbolj ustrezna je RF (radio frequency) komunikacija, ker nudi dovolj velik doseg in hitrost prenosa podatkov. Pomembno je tudi, da ni potrebno, da sta elementa, ki komunicirata v vidnem polju. Oddajnik mora prevesti digitalno zapisane podatke v obliko radijskih valov, sprejemnik pa ravno obratno. Omeniti velja, da preprosti oddajniki in sprejemniki, ki so komponente elementov BSO večinoma nimajo enotnega identifikatorja, po katerem bi se elementi med seboj razlikovali. Identifikator bi pomenil zelo visoko ceno posameznega oddajnika in sprejemnika v primerjavi s ceno celotnega elementa.

Ker za izgradnjo BSO potrebujemo veliko število elementov, je pomembna tudi cena. Seveda je za konkretne primere uporabe potrebno skleniti nekaj kompromisov, za nekatere aplikacije morda poraba energije ne bo tako pomembna kot cena celotnega omrežja.

Ne nazadnje sta pomembni tudi velikost in oblika, odvisno od namena uporabe. Velikost vezij in s tem njihova poraba se je od izuma integriranega vezja leta 1958 zmanjševala in s tem omogočila tudi razvoj elementov, ustreznih za BSO.

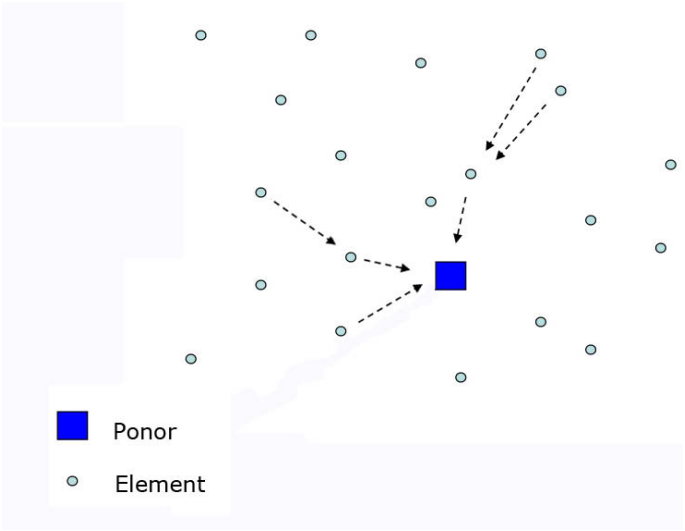
Omenimo še, da ni nujno, da je omrežje sestavljeno iz povsem enakih elementov. Elementi se med seboj lahko razlikujejo, pomembno je le, da je med njimi mogoče vzpostaviti komunikacijo. Primer omrežja, ki je sestavljeno iz različnih elementov je lahko tekaška oprema, kjer različni elementi opravljajo različne meritve.

Podrobneje bomo predstavili element Zolertia Z1 v poglavju 3.

2.2 Povezava z okoljem in zajemanje podatkov

BSO ni le brezžično omrežje za prenos podatkov. Izredno pomembno je zajemanje podatkov iz okolice in posledično tudi njihov prenos do ponora. Element omrežja, opremljen s senzorjem, je izvor podatkov v BSO, ponor pa je lahko prav tako element, ali pa naprava, ki ni del omrežja. Zelo enostaven primer BSO je lahko "tekaška oprema": tekaški copat z vgrajenim senzorjem za merjenje števila korakov (prvi element omrežja), pas s senzorjem za merjenje srčnega utripa (drugi element omrežja) in ura, na kateri lahko spremljamo meritve obeh elementov omrežja (ponor). Sama komunikacija med omenjenimi tremi elementi "tekaške opreme"uporabniku ne ponuja ničesar, zanimajo ga podatki, ki jih lahko med tekom spremlja.

Tako izvorov kot tudi ponorov imamo lahko v BSO več. Merjenje povprečne temperature na stadionu lahko zagotovimo z BSO, ki je sestavljen iz več elementov in enega ponora (slika 2.3) v obliki na primer mobilnega telefona. Spremljanje gozdnega požara s senzorji, ki smo jih raztresli po gozdu pa podajmo kot primer BSO, ki ima več ponorov (slika 2.4) – gasilci se žarišču požara bližajo z različnih smeri in vsak ima napravo (npr. pametni telefon), ki predstavlja ponor, s katero spremlja temperature tal. Tipično v literaturi

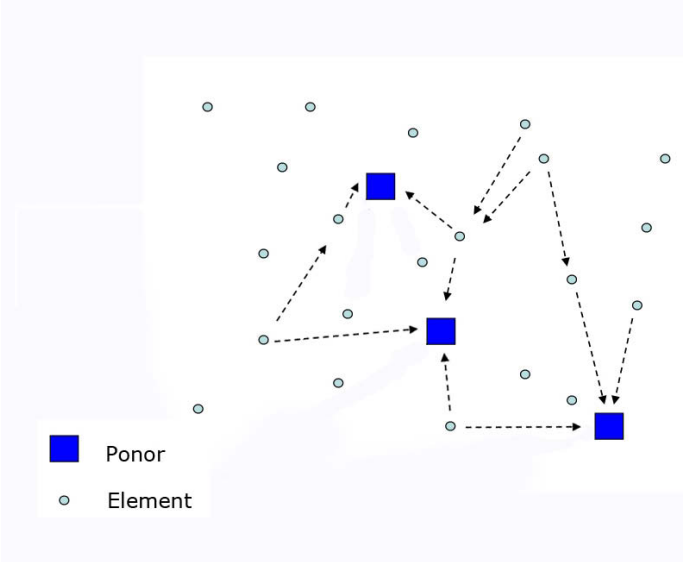


Slika 2.3: Shematski prikaz BSO z enim ponorom in večimi izvori, vir [2]

srečamo BSO z enim ponorom in več izvori.

S pomočjo senzorjev zbiramo in obdelujemo podatke iz okolice, ki nas zanima. Velja omeniti, da ni pravila, ali podatke obdelujemo šele ko jih zberemo, ali nam jih obdela že omrežje. V kolikor želimo pridobiti že obdelane podatke (na primer povprečno temperaturo) to lahko storimo s sodelovanjem med elementi omrežja. Le-ti lahko prejet paket združijo s podatkom prebranim s senzorja in v omrežje naprej namesto dveh (prejet paket sosednjega elementa in paket s podatki elementa), pošljejo le en paket. Združevanje paketov je izredno koristno zaradi zmanjševanja števila paketov, ki jih prenašamo po omrežju. Sprejemnik in oddajnik elementa BSO je večinoma največji porabnik električne energije, dlje časa kot je element aktiven v smislu pošiljanja ali prejemanja paketov, več energije potroši. S tem se lahko skrajša življenjska doba celotnega omrežja.

Podatke lahko v omrežje pridobimo na več načinov, najpogostejša pa sta detekcija dogodkov (senzor nam sporoči, ali se je dogodek zgodil) in periodične meritve (temperaturo merimo v določenem časovnem intervalu). Dogajanje, ki ga spremljamo, vpliva na omrežje, njegovo strukturo (omrežje

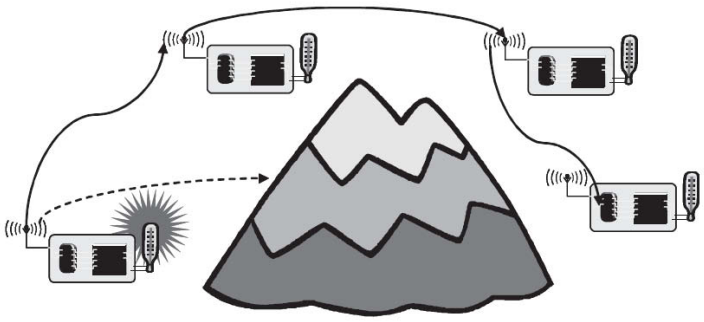


Slika 2.4: Shematski prikaz BSO z več ponori in več izvori, vir [2]

lahko prilagodimo našim zahtevam) in na količino prenesenih podatkov po omrežju. To pa je tudi ena izmed glavnih razlik, po katerih ločimo BSO od ostalih brezžičnih omrežij, kjer je pomemben le prenos podatkov.

2.3 Energijska učinkovitost in robustnost BSO

Za učinkovito delovanje BSO moramo imeti vzpostavljeno ustrezno komunikacijo med elementi. Da lahko od določenega elementa omrežja prejmemo želene podatke, morajo le-ti prispeti do ponora, kjer jih zbiramo. Domet omrežja in izbira načina komunikacije (protokola) predstavlja eno od težav načrtovanja BSO, saj za pošiljanje sporočil daleč potrebujemo zelo veliko energije, to pa skrajšuje življensko dobo omrežja. Težave se lahko pojavijo tudi zaradi terena, kjer imamo razporejene elemente. Omrežje je lahko v visokogorju, kjer signal motijo visoki hribi (slika 2.5), v urbanem okolju, kjer predstavljajo ovire zgradbe ali druge motnje brezžične komunikacije, ipd. V večini primerov je nemogoča neposredna povezava (prenos podatkov) od izvora do ponora in potrebujemo mehanizme, ki omogočajo pošiljanje sporočil



Slika 2.5: Shematski prikaz težav prenosa podatkov zaradi terena, vir [1]

preko vmesnih elementov.

Izbira načina komunikacije je v BSO zelo pomembna za optimizacijo delovanja in predvsem energijsko učinkovitost. V primeru napačno izbranega načina prenosa podatkov po omrežju lahko zaradi nepotrebne komunikacije med elementi zelo hitro iztrošimo baterije in s tem omrežje postane neuporabno.

Zapisali smo že, da zaradi narave omrežja elementi večinoma ne morejo biti priključeni na zunanje vire energije in se zanašajo na baterije, kar pomeni, da je energijska učinkovitost takega omrežja zelo pomembna. Za ilustracijo pomena energijske učinkovitosti podajmo primer uporabe - vzemimo opazovanje morskega dna. Elemente razporedimo po morskem dnu in s pomočjo senzorjev spremljamo erozijo morskih tal, do katere pride zaradi morskih tokov in valov. V takem težko dostopnem okolju nimamo možnosti menjave baterij ali elementov in pričakujemo dolgotrajno delovanje omrežja brez večjih posegov. Dolgotrajno delovanje moramo razumeti v primeru BSO kot večletno neprekinjeno delovanje, ali vsaj večmesečno delovanje, ne da bi bili v omrežju potrebni večji posegi.

Zaradi zahteve po več let trajajoči življenjski dobi je izredno pomembna tudi robustnost omrežja. Življenjska doba omrežja je po najbolj splošni definiciji časovni interval, ki se začne s prvim brezžičnim prenosom podatkov po

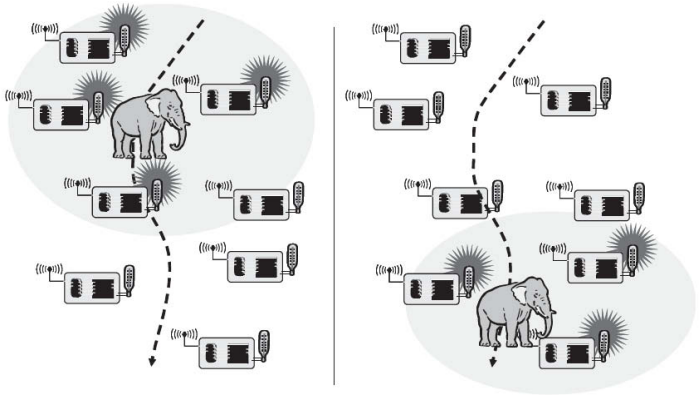
omrežju in konča, ko odstotek elementov, ki imajo še dovolj energije za oddajanje in sprejemanje, pade pod določen prag [2]. To število določimo glede na način uporabe – lahko že odpoved enega elementa pomeni konec življenjske dobe omrežja. Vendar težimo k temu, da je omrežje čimbolj neobčutljivo na izpade posameznih elementov, izgubo paketov ali morebitne neugodne vplive okolja. V primeru izpada enega ali več elementov mora biti enostavno nadomestiti primankljaj, bodisi z novimi elementi ali z možnostjo, da že obstoječi elementi omrežja prevzamejo vlogo nedelujočega. Omrežje mora biti dovolj robustno in odporno na napake. V določenih primerih lahko od BSO pričakujemo tudi sposobnost rekonfiguracije brez zunanjega vpliva.

Koncept BSO nam nudi zelo veliko število možnih načinov izgradnje omrežja (neomejeno število elementov, brezžična povezava s prosto izbiro načina komunikacije, avtonomnost, zaznavanje različnih količin iz okolja, možnost vplivanja na okolje, dolga življenjska doba, različna strojna oprema). Zato je zelo pomembno, da vnaprej točno definiramo cilj, ki ga želimo doseči s pomočjo BSO in s tem omogočimo optimizacijo omrežja, dobro energijsko učinkovitost in s tem večletno življensko dobo.

2.4 Mobilno ad hoc omrežje in BSO

Ker smo večkrat poudarili razliko med BSO ter ostalimi brezžičnimi omrežji, podajmo še konkretnejši primer s krajšo primerjavo. Mobilno ad hoc omrežje ali MANET (angl. Mobile ad hoc network) je avtonomno omrežje sestavljeno iz premikajočih se elementov, brez infrastrukture, povezanih brezžično. Primer enostavnega MANET omrežja so lahko prenosni računalniki v konferenčni dvorani povezani med seboj zaradi izmenjave podatkov – računalniki se lahko premikajo, enega ali več jih lahko izklopimo, ostali si bodo še vedno lahko izmenjevali podatke.

MANET uporablja močnejše procesorje, več pomnilnka, kar pomeni da ni energijsko učinkovito kot si to želimo za BSO. V nasprotju z MANET se BSO lahko popolnoma spremeni odvisno od problema, a MANET ostaja enak in



Slika 2.6: Spremljanje dogodka z BSO, vir [1]

z okoljem ne sodeluje. Povezava točka-točka za BSO ni najbolj optimalna, medtem ko pri MANET to ni problem. Moč oddajanja je odvisna od oddaljenosti posameznih elementov od ponora in za prenos podatkov bi porabili preveč energije z večjo oddajno močjo. Zato potrebujemo ustrezne mehanizme, ki omogočijo usmerjanje paketov po omrežju preko ostalih elementov. Več o protokolih in komunikaciji v BSO je zapisano v knjigi [1].

2.5 Mobilnost BSO

Mobilnost BSO je za razliko od klasičnih omrežij lahko zelo kompleksna. Možni so premiki elementov omrežja, lahko spremljamo dogodek, ki se premika (npr. sledenje divjim živalim, slika 2.6), premika se lahko ponor (npr. naprava, ki jo nosi oseba). Tudi to lahko predstavlja težavo pri izbiri načina komunikacije.

2.6 Varnost BSO

Omenimo še varnost BSO, ki je ena izmed perečih težav takšnih omrežij. Ker zahtevamo nizko porabo energije in imamo na voljo le malo virov, je uporaba zapletenih kriptografskih algoritmov skoraj nemogoča. Mogoč je

tudi fizičen dostop do senzorjev in možnost motenja radijskega signala na območju omrežja, kar pomeni dodatno težavo, s katerimi se v takem smislu v klasičnih omrežjih načeloma ne srečujemo. Več o varnosti omrežja v [1] in [6].

Če povzamemo do sedaj zapisane lastnosti BSO in elementov omrežja, lahko zapišemo, da gre za omrežje sestavljeno iz majhnih, enostavnih in energijsko učinkovitih elementov, s katerimi zaznavamo okolje in nanj lahko tudi vplivamo.

Poglavje 3

Element Zolertia Z1

Zolertia Z1, prikazan na sliki 3.1, je učni, študijski in testni element BSO. Namenjen je gradnji in razvoju prototipnega brezžičnega senzorskega omrežja.

Slika 3.1: Element Zolertia Z1 v ohišju z zunanjo anteno, kovanec za primerjavo velikosti

Po specifikacijah proizvajalca je bil element omrežja razvit s sledečimi lastnostmi [9].

- Platforma za hitro izdelavo prototipov in izgradnjo BSO
- Deluje v razponu temperatur, ki so primerne za industrijsko uporabo (-40^oC do 85^oC)

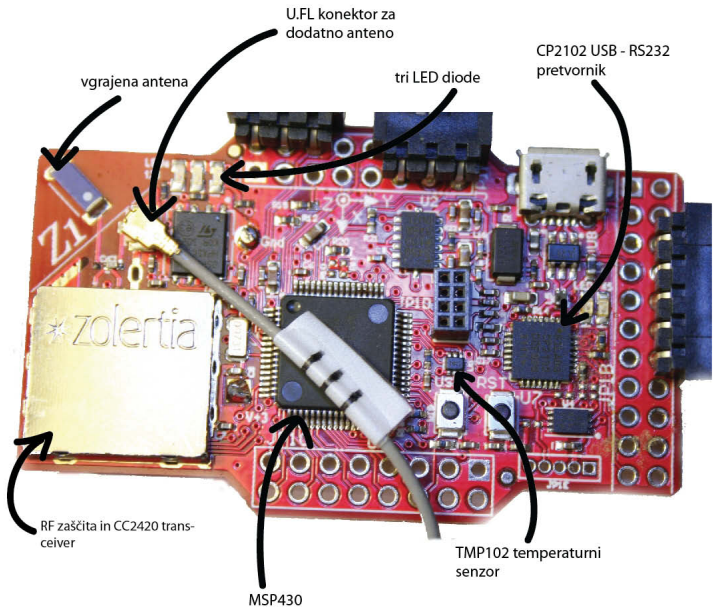
- Ima 52-pinski razširitveni konektor
- Zgrajen je na osnovi druge generacije mikrokontrolerja Texas Instruments MSP430
- Vsebuje oddajnik in sprejemnik CC2420
- Vgrajena ima dva senzorja: temperaturni senzor (TMP102) in pospeškomer (ADXL345)
- Priključimo lahko dodatno do štiri zunanje senzorje in zunanjo anteno
- Za programiranje imamo na voljo mikroUSB priključek

3.1 Oddajnik in sprejemnik

Integrirano vezje CC2420 [8] je oddajnik in sprejemnik na enem čipu, ki ustreza standardu IEEE 802.15.4 in se uporablja v brezžičnih omrežjih, kjer je pomembna nizka poraba energije. Omogoča hitrost prenosa podatkov 250 kbps (kilobit per second). Zaradi zagotavljanja nizke porabe električne energije uporablja *enegry detection*. To pomeni, da ima vgrajen RSSI (Received Signal Strength Indicator), kar nam omogoča meritev moči signala. V telekomunikacijah je RSSI definiran kot količina moči (ponavadi izmerjena v dBm), ki je prisotna v prejetem signalu. Na podlagi moči signala lahko sklepamo na kvaliteto povezave med elementi in do neke mere natančno tudi ocenimo razdaljo med njimi. Tokovna poraba CC2420 je pri oddajanju $17.4mA$ (z močjo $0dBm$ ali $1mW$), pri sprejemanju pa $18.8mA$.

3.2 Napajanje

Napajanje Z1 je možno na več načinov, odvisno od načina uporabe samega elementa v omrežju. Najpogostejša načina napajanja sta z dvema baterijama velikosti AA ali preko mikroUSB priključka. Možno je tudi napajanje s CR2032 baterijo, za katero je prostor na spodnji strani vezja, vendar je v tem primeru potrebno še posebno pazljivo razvijati programsko opremo z mislijo ohranjanja električne energije.



Slika 3.2: Lokacija temperaturnega senzorja TMP102 ter ostalih komponent na elementu Z1

Dejanska splošna izmerjena poraba energije na bateriji elementa Z1 je bila med izvajanjem aplikacije za merjenje temperature približno $0,7mA$, med pošiljanjem podatkov pa $10mA$. Iz tega lahko približno izračunamo, da bo življenjska doba elementa Z1 z dvema AA baterijama približno tri mesece pri uporabi naše aplikacije.

3.3 Temperaturni senzor TMP102

Element Z1 ima vgrajena dva senzorja in sicer temperaturni senzor (TMP102) in pospeškomer (ADXL345). Temperaturni senzor TMP102 proizvajalca Texas Instruments je za realizacijo našega končnega sistema ključnega pomena. Temperaturni senzor je vsebovan v integriranem vezju TMP102 in za merjenje temperature ne potrebuje dodatnih komponent. Senzor ima resolucijo temperature do $0,0625$ stopinje Celzija.

Lokacija TMP102 na elementu Z1 je prikazana na sliki 3.2.

Pri zajemanju temperature moramo paziti na morebitne napake v meritvah (do $+2°C$), v kolikor je senzor, ki opravlja meritev, priklopljen preko USB priključka na strežnik. Do napake pride zaradi segrevanja komponente CP2102 (pretvornik) elementa, ki se na vezju nahaja v neposredni bližini temperaturnega senzorja TMP102. To je razvidno tudi iz slike 3.2.

3.3.1 Osnovne tehnične specifikacije senzorja

Osnovne tehnične specifikacije senzorja TMP102 [10]:

- Majhno SOT563 ohišje
- Natančnost meritev: $0.5°C$ ($-25°C$ do $+85°C$)
- Tok: $10\mu A$ Active (max), $1\mu A$ Shutdown (max)
- Napetost napajanja: najmanj $1.4V$ do največ $3.6V$
- Vodilo: I2C (Two-Wire Serial Interface)

3.3.2 Registri temperaturnega senzorja TMP102

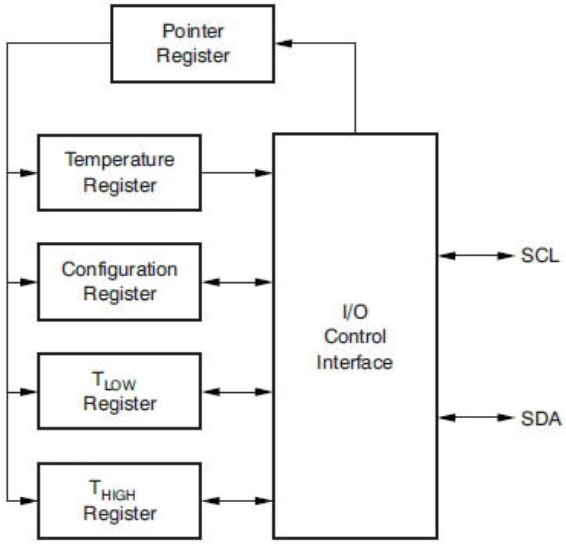
TMP102 ima 5 registrov (slika 3.3). Poleg temperaturnega registra, kjer je shranjena izmerjena temperatura, imamo še 4 registre, ki nam omogočajo poenostavitev programske opreme za delo s senzorjem. Ti registri so pointer ali kazalni register, konfiguracijski register in High ter Low registra. V nadaljevanju so predstavljene funkcije teh registrov.

Pointer ali kazalni register

Pointer ali "kazalni"register je 8-bitni register kazalcev temperaturnega senzorja in ga uporabimo za naslavljanje podatkovnih registrov. Pri pisanju so biti registra od 2 do 7 postavljeni na 0, bita 0 in 1 pa določata, do katerega izmed ostalih 4 registrov dostopamo.

Konfiguracijski register

Konfiguracijski register je 16-bitni bralno pisalni (R/W) register za kontrolne bite, ki določajo načine delovanja. Omenimo bita EM ter TM. Bit EM določa



Slika 3.3: Registri temperaturnega senzorja TMP102, vir [10]

velikost temperaturnega registra. V kolikor je bit EM postavljen na 0 uporabljamo 12 bitni register, v nasprotnem primeru pa 13 bitnega. Bit TM določa, ali je senzor v primerjalnem načinu ali ne. V kolikor je postavljen na 1 je senzor v primerjalnem načinu in lahko uporabimo registra High in Low limit, ki sta opisana v nadaljevanju.

Temperaturni register

V temperaturnem registru je shranjena v binarni obliki zapisana izmerjena temperatura v $°C$. Register je lahko 12-bitni ali 13-bitni, odvisno od bita EM v konfiguracijskem registru. Register je bralni, kar pomeni, da ne dovoljuje pisanja. Hkrati preberemo 2 bajta, pri čemer ima prvi bajt večjo težo. Bit z najnižjo težo ima vrednost 2^{-4} $°C$ kar ustreza $0,0625$ $°C$.

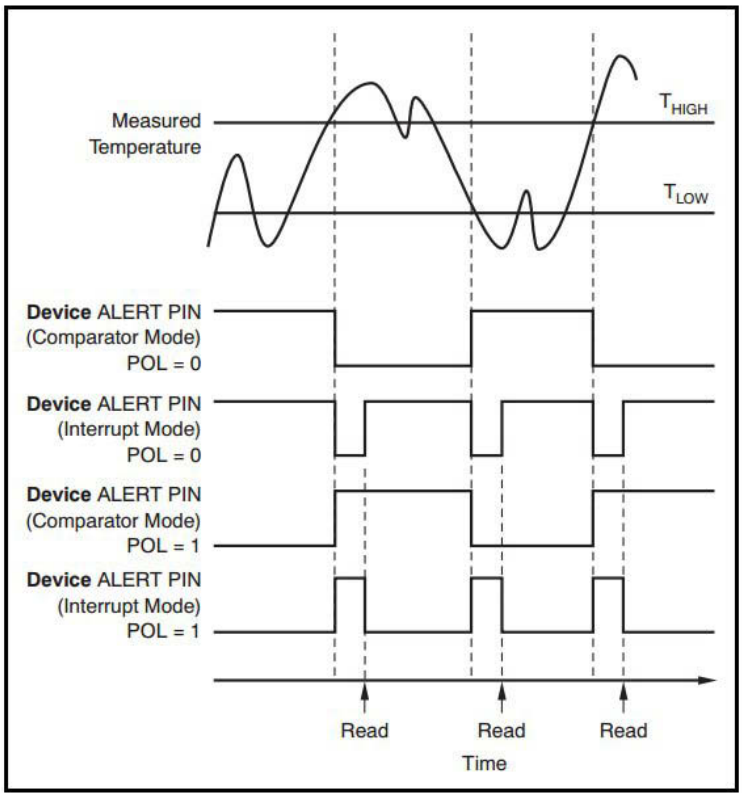
Negativna števila so v registru predstavljena z dvojiškim komplementom. Dvojiški komplement dobimo tako, da eniškemu komplementu števila prištejemo 1. Zadnji bit registra se postavi glede na bit EM v konfiguracijskem registru. Ostali neuporabljeni biti so vedno 0.

Podajmo primer zapisa v temperaturnem registru za pozitivno število:

$50 = 110010_{(2)}$, bit z največjo težo je 0 (pozitiven predznak), predzadnji 3 biti so nerabljeni in posledično 0, zadnji pa prav tako (12 bitni zapis, bit EM = 0). Tako dobimo zapis v registru 001100100000.

In še primer zapisa v registru za negativno število:
$-25 = 111001101111_{(2)}$ (ker je $25 = 11001_{(2)}$, oziroma v registru 12- bitno 000110010000) $+1 = 111001110000$.

High in Low limit registra



Slika 3.4: Uporaba registrov T_{HIGH} in T_{LOW} TMP102, vir [10]

T_{HIGH} in T_{LOW} limit registra imata enak format zapisa kot Temperaturni register. Uporabimo ju v primeru, da želimo prožiti akcijo na podlagi primerjave vrednosti, ki jo shranimo v registra. Če je TMP102 v primerjal-

nem načinu (kontrolni bit TM = 0) se opozorilo sproži (ALERT pin postane aktiven), ko izmerjena temperatura doseže ali preseže vrednost v T_{HIGH} registru. ALERT pin je aktiven v visokem ali nizkem stanju - to določimo s POL bitom. ALERT pin ostane aktiven dokler teperatura ne pade pod vrednost zapisano v T_{LOW}. Če je senzor v "primerjalnem načinu"to pomeni, da je kontrolni bit TM = 1 in se opozorilo sproži, ko temperatura preseže ali doseže vrednost v T_{HIGH}. Opozorilo sprostimo, ko preberemo enega izmed registrov. Ponovno se bo sprožilo, ko temperatura pade pod vrednost zapisano v registru T_{LOW}. Grafično je proženje opozorila z uporabo registrov T_{HIGH} in T_{LOW} prikazano na sliki 3.4).

Z uporabo teh dveh registrov lahko bistveno zmanjšamo porabo energije, kar je za elemente BSO ključnega pomena. Če želimo poslati temperaturo le v primeru, ko ta preseže ali pade pod določeno vrednost, zbudimo element le takrat. S pomočjo opozorila zbudimo element, ki je sicer med periodičnim merjenjem temperature v stanju mirovanja. Pošljemo temperaturo, nato pa ponovno postavimo element v stanje mirovanja.

Primer branja temperaturnega registra

Kako bomo brali temperaturo registra je odvisno tudi od gonilnikov, ki so za določen element na voljo. V nadaljevanju je podan primer branja temperature iz registra TMP102 in izpis v stopinjah celzija v operacijskem sistemu Contiki [7].

```
sign = 1; //predznak
// Branje registra senzorja
raw = tmp102_read_temp_raw();

absraw = raw;
// Dvojiski komplement, negativno stevilo
if (raw < 0) {
  absraw = (raw ^ 0xFFFF) + 1;
  sign = -1;
}
//celi stevilski del
tempint  = (absraw >> 8) * sign;
// 1/10000 stopinje, za decimalno vejico
tempfrac = ((absraw>>4) % 16) * 625;
minus = ((tempint == 0) & (sign == -1)) ? '-'  : ' ' ;
//izpis
printf ("Temp = %c%d.%04d\n", minus, tempint, tempfrac);
```

Klic funkcije *tmp102_read_temp_raw()* nam vrne vrednost temperaturnega registra v "raw"obliki. V kolikor bi želeli izpisati le celoštevilski del na stopinjo natančno, uporabimo funkcijo *tmp102_read_temp_simple()*. Device driver header za temperaturni senzor TMP102 za operacijski sistem Contiki za element Zolertia Z1 je dostopen na *http://dak664.github.io/contiki-doxygen/a01543_source.html*.

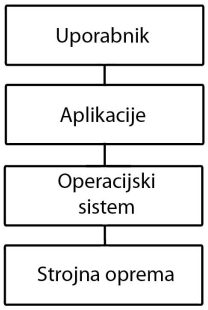
Poglavje 4

Operacijski sistemi za brezžična senzorska omrežja

V tem poglavju bomo predstavili nekaj osnovnih značilnosti vgrajenih (angl. embedded) operacijskih sistemov in operacijskih sistemov za BSO. Ker se programiranje aplikacij za BSO nekoliko razlikuje od klasičnega, se bomo dotaknili tudi te teme. V nadaljevanju bomo predstavili dva odprtokodna operacijska sistema za brezžična senzorska omrežja, ki sta trenutno (2014) najbolj pogosto omenjana v literaturi: TinyOS in Contiki. V praktičnem delu bomo vzpostavili delovno okolje za delo z obema operacijskima sistemoma in v zadnjem delu poglavja oba primerjali. Omenili bomo tudi nekaj drugih operacijskih sistemov, ki so na voljo za delo z BSO.

4.1 Operacijski sistem

Operacijski sistem (v nadaljevanju OS) je programska oprema, ki upravlja s strojno opremo računalniškega sistema in nadzira delovanje celotnega računalnika. Operacijski sistem mora poskrbeti, da vsi drugi programi delujejo pravilno in s tem zagotavljati storitve uporabniku sistema. Lahko rečemo, da gre za neke vrste vmesnik med programi (aplikacijami) in strojno opremo. Za nemoteno delovanje programov, ki jih uporabnik računalnika izvaja, na primer ureje-



Slika 4.1: Shema za prikaz umeščenosti operacijskega sistema v računalniškem sistemu

valnik besedil, brskalnik ali računalniška igra, skrbi OS. Shematični prikaz umeščenosti OS v računalniškem sistemu je prikazan na sliki 4.1.

4.2 Primerjava strojne opreme

Za boljše razumevanje omejitev, ki jih predstavlja izdelava operacijskega sistema namenjenega BSO, si oglejmo primerjalno tabelo strojne opreme (tabela 4.1). Klasične operacijske sisteme, ki so namenjeni širši uporabi (pod te štejemo Windows, Unix-like, Mac OS) večinoma poganjajo splošno namenski osebni računalniki. Zato smo za primerjavo vzeli povprečni osebni računalnik, ki je danes (2014) v prodaji. Povprečni računalnik je računalnik srednjega cenovnega razreda. Kot primer strojne opreme, primerne za BSO, pa smo izbrali element Zolertia Z1.

Operacijski sistem mora ustrezati strojni opremi. Iz tabele 4.1 je razvidno, da mora biti operacijski sistem za BSO podrejen cilju majhne porabe energije in virov, ki jih imamo na voljo. V primerjavi z osebnim računalnikom imamo na voljo kar 10^6 -krat manj glavnega pomnilnika in 10^8 -krat manj zunanjega pomnilnika. Zato moramo paziti na velikost operacijskega sistema in ostale programske opreme, ter na količino porabljenega zunanjega pomnilnika. Tudi hitrost in zmogljivost procesorja je veliko večja v osebnem

	Osebni računalnik	**Zolertia Z1**
procesor	Intel Core i5 4440 64-bit (3,1 GHz)	TI MSP430f2617 16-bit mikrokontroler (0.016GHz)
glavni pomnilnik	8 GB (8388608 KB)	8 KB pomnilnika v MSP430
zunanji pomnilnik	trdi disk 1 TB (10^9 KB)	92 KB flash pomnilnika v MSP430
napajanje	napajalnik priključen na električno omrežje	z baterijami ali preko mikro-USB

Tabela 4.1: Primerjalna tabela strojne opreme

računalniku. Predvsem pa smo omejeni z napajanjem in operacijski sistem za BSO mora zato omogočati čimboljši izkoristek energije, ki je na voljo.

4.3 Vgrajeni operacijski sistem

Vgrajeni operacijski sistemi (v nadaljevanju VOS) so v splošnem sistemi, ki izvajajo specifične zahteve in točno določene naloge na vgrajeni strojni opremi. VOS za BSO morajo biti zasnovani kompaktno, zmogljivo za omejene vire, robustno in zanesljivo, pri čemer se morajo odreči večini funkcionalnosti, ki jih sicer srečamo pri operacijskih sistemih.

Strojna oprema, ki jo uporabimo za BSO ima v večini primerov na voljo zelo malo pomnilnika, kar pomeni, da morajo biti naloge sistema znane vnaprej, da lahko omejen pomnilnik izkoristimo za izbrana opravila.

Operacijski sistem v splošnem nadzira porabo vseh virov, ki jih ima računalniški sistem na voljo in njihov čas razporeja med porabnike, skrbi za sočasno izvajanje več procesov in komunikacijo med njimi. V BSO večinoma teh nalog ne potrebujemo, saj je tudi programska koda in s tem procesi omejena na maloštevilne vire. Zaradi strojne opeme pa je tudi povsem nesmiselno razvijati klasičen operacijski sistem, saj mikrokontrolerji, na katerih je osnovanih veliko vgrajenih sistemov (in elementov BSO) niso dovolj zmogljivi.

Za VOS BSO zahtevamo, da je prilagojen osnovnim splošnim zahtevam za BSO:

- v BSO nas zanimajo dogodki, odgovor na določeno "vprašanje"v pravem trenutku, zato potrebujemo podporo za izvajanje programov v realnem času
- prenos podatkov ne sme povzročiti prevelike porabe energije in po nepotrebnem krajšati življenjske dobe BSO
- omrežje mora biti robustno, odporno na napake (na primer morebitno odpoved vozlišča) in do neke mere sposobno rekonfiguracije brez zunanjih vplivov
- življenjska doba posameznih vozlišč mora biti čim daljša, zelo pomembna je energetska učinkovitost celotnega BSO
- razporeditev vozlišč v omrežju je odvisna od namena uporabe in je lahko povsem različna
- mogoča mora biti uporaba različnih načinov komunijacije
- podatki o dogodkih se lahko delno obdelajo že v omrežju
- omogočeno mora biti sodelovanje med vozlišči v kolikor to zahteva problem, ki ga rešujemo

Operacijski sistem za BSO mora biti neodvisen od strojne opreme. Izbira elementa in njegovih komponent je odvisna od namena uporabe in mogoče je, da bomo za dva BSO z različnima namenoma uporabe uporabili različne elemente.

Predvsem pa je pomembna energijska učinkovitost VOS, za kar potrebujemo učinkovit sistem za upravljanje z energijskimi viri.

Vse komponente elementa BSO zahtevajo zmogljivo in preprosto rabo. Ker lahko iz okolja kadarkoli dobimo dražljaj, ki ga je potrebno z omrežjem zaznati in ustrezno odreagirati, moramo imeti mehanizme, ki to omogočajo. Obenem pa ne smemo dopustiti, da bi katerakoli od komponent elementa BSO delovala dlje ali z večjo močjo, kot je nujno potrebno. Če želimo periodično meriti temperaturo, zbudimo element omrežja le ob ustreznem času. Nato izmerimo temperaturo in jo posredujemo naprej, zatem pa ponovno element postavimo v stanje mirovanja. Element v času, ko ne merimo temperature ni v aktivnem stanju in porabi le toliko energije, kolikor je nujno

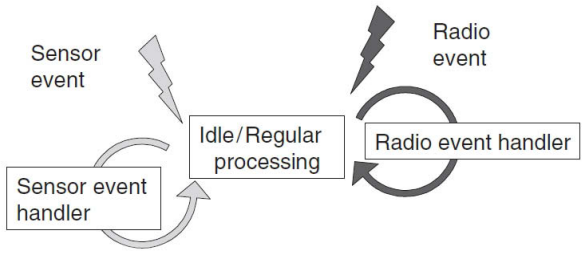
potrebno. Podobno velja za detekcijo dogodkov. V času, ko čakamo na dogodek, mora biti element v mirovanju, nato pa ga moramo ustrezno zbuditi, da opravimo zahtevane operacije in element ponovno postaviti v stanje mirovanja.

Kot za druge OS tudi za VOS za BSO velja, da je pomembno, da se razvijalci programske opreme, ki jo razvijajo za določen sistem, ne ukvarjajo toliko s strojno opremo, kjer bo program nameščen, temveč s programom, ki ga razvijajo. Za povezovanje s strojno opremo mora poskrbeti OS. Kljub temu se programiranje za BSO nekoliko razlikuje od "klasičnega"programiranja.

4.4 Programiranje za BSO

Pri programiranju za BSO se najprej soočimo s problemom, kako zagotoviti vzporednost. Možen je splet okoliščin, ko v poljubnem času zaznamo več dogodkov hkrati, preko različnih senzorjev. Istočasno lahko že obdelujemo nek drug podatek. Zato moramo zagotoviti vzporednost. Kot primer podajmo BSO, s katerimi želimo zaznati požar v določeni zgradbi. Tako omrežje mora zaznati več morebitnih požarov hkrati in na vseh lokacijah, kjer gori, vklopiti gasilne aparate. Če bi omrežje opozorilo le na en požar, a bi zagorelo v več nadstropjih hkrati, to ne zagotavlja učinkovite protipožarne varnosti. Iz primera je razvidno, da enostavni, sekvenčni model programiranja ne zadostuje.

Večina operacijskih sistemov danes podpira (navidezno) paralelno izvajanje procesov znotraj enega procesorja. V teoriji bi lahko tak koncept prenesli tudi na BSO, vendar se pokaže, da to ni smiselno [1]. Že zato, ker vsak proces zahteva svoj delež pomnilnika, ki ga nimamo dovolj na voljo, vidimo, da moramo uporabiti drugačen pristop.



Slika 4.2: Model programiranja pogojenega z dogodkom, vir [1]

4.4.1 Programiranje pogojeno z dogodkom[1]

Potek programa je v takem načinu programiranja odvisen od dogodkov v okolici, ki jih spremljamo. Dogodek je lahko premik določene živali v nek predel gozda, dvig temperature prostora, sprememba lokacije ali sprejem ustreznega radijskega signala. V splošnem ima tak program glavno zanko, ki čaka na dogodek. Ustrezen dogodek nato (s)proži določeno funkcijo (slika 4.2). Izkaže se, da je tak način programiranja elementov BSO najbolj primeren [1].

V primeru ustreznega dogodka, na katerega smo čakali, sprožimo obdelovalnik (angl. handler) dogodka, ki shrani nekaj osnovnih informacij o dogodku.

Obdelovalniki dogodka so v večini primerov programske procedure ali funkcije, ki opravijo določeno akcijo in se nato vrnejo h klicatelju. Ker jih ne moremo prekiniti in se izvedejo v celoti, obdelovalnik dogodka ne more čakati.

Obdelava dobljenih podatkov se nato zgodi nepovezano s pojavom dogodka. Obdelovalnik dogodka lahko prekine izvajanje druge programske kode, vendar zahtevamo, da se prekinitrev izvede hitro in da ne prekine drugega obdelovalnika dogodka, ki mora čakati na izvedbo prvega in sicer po principu FIFO[2]. Razlikujemo med obdelovalniki dogodka, ki jih ne smemo

[1] V angleški strokovni literaturi se uporabljata izraza "event based"ali "event driven programming", ki ju bomo prevajali kot programiranje pogojeno z dogodkom

[2] angl. First In, First Out: prva zahteva, ki pride v vrsto gre prva tudi ven

(ne moremo) prekiniti in med ostalo programsko kodo, ki jo obdelovalniki sprožijo.

4.4.2 Protothreads

Ker je zaradi omenjenih zahtev programiranje v sistemu, ki je pogojen dogodkom zahtevno opravilo, so snovalci operacijskega sistema Contki (ki ga bomo podrobneje predstavili v poglavju 4.6) predstavili idejo "protonitnosti"(angl. protothreads) [12]. S tem programerskim konceptom naj bi olajšali kompleksnost programiranja za BSO, saj pri pisanju kode uporabimo standardne zanke (while) in pogojne stavke (if). Osnovna ideja "protonitnosti"je v dodajanju dodatnega klica, ki skrbi za preklapljanje opravil v vse zanke.

Tako pridobimo strukturo, ki je, lahko rečemo, nekje vmes med polno večnitnostjo in z dogodkom pogojenim načinom programiranja. Na voljo imamo tako blokiranje in čakanje procesov[3], vendar ne potrebujemo velikega števila skladov.

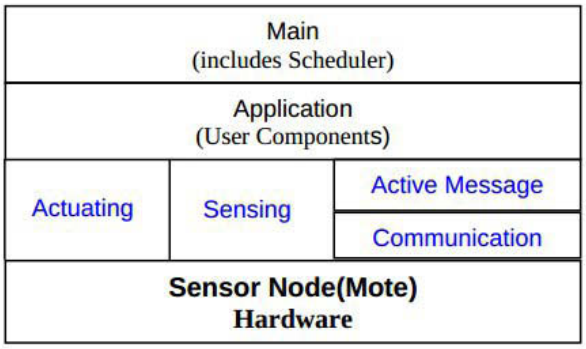
Za vsako "protonitnost"operacijski sistem Contiki porabi dva bajta in nobenih dodatnih skladov. Ker so v celoti zapisani v programskem jeziku C, protonitnosti niso odvisne od arhitekture sistema. Več o protonitnosti je dostopno na [15] in [12].

4.5 TinyOS

TinyOS je prost odprtokoden operacijski sistem izdan pod licenco New BSD. Prvotno je namenjen uporabi v BSO, lahko pa ga uporabimo tudi v ostalih sistemih, kjer je pomembna nizka poraba energije in pomnilniškega prostora. TinyOS podpira veliko različnih elementov [13], med drugim MicaZ, TMote Sky, TinyNode, Zolertia Z1 in UCMote Mini.

Programska koda TinyOS je zapisana v programskem jeziku nesC, ki je razširitev programskega jezika C. nesC prevajalnik je napisan v programskem

[3]Proces, ki je "blokiran"je proces, ki čaka na nek dogodek, kot je naprimer sprostitev virov.



Slika 4.3: Poenostavljen shematični prikaz arhitekture TinyOS, vir [17]

jeziku C. Jezik nesC je prilagojen omejitvam, ki jih imajo brezžična senzorska omrežja. Nekoliko podrobneje ga bomo predstavili v nadaljevanju.

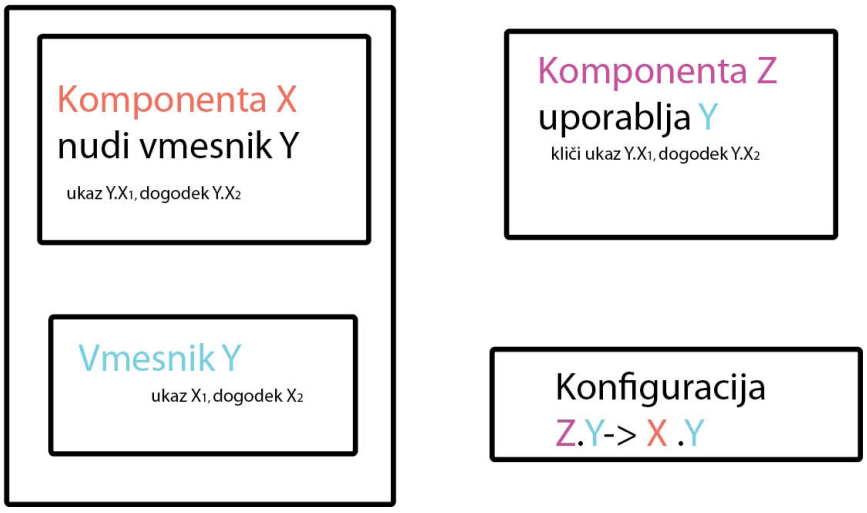
TinyOS je zasnovan s tako imenovano monolitsko arhitekturo (slika 4.3). To pomeni, da funkcionalnosti sistema, ki se sicer razlikujejo (podatkovni vhodi, procesiranje podatkov ali odpravljanje napak) niso dejansko svoje komponente. Povedano drugače, v monolitskih sistemih se program prevede skupaj z operacijskim sistemom kot "en program". Ima en sklad, ki si ga delijo vse komponente. Pomnilnik, ki je na voljo uporabniku je na voljo tudi jedru sistema.

TinyOS na višjem nivoju nudi uporabniku [4]:

- komponentni model,
- model za sočasno izvajanje in
- API (Application programming interface).

Komponentni model[4] omogoča večkratno uporabo že spisane programske kode. Komponente so razporejene hierarhično, od nižje nivojskih, ki so bližje strojni opremi, do visoko nivojskih, ki so sestavni del aplikacij. Dogodki "izhajajo"iz najnižjega nivoja in se morajo prenesti v smeri proti komponentam višjega nivoja, klici oziroma ukazi pa se prenašajo v obratni smeri. Komponente so tako povezane v obe smeri. Razumemo jih lahko tudi kot delitev

[4]Component-based software engineering



Slika 4.4: Primer aplikacije TinyOS, povzeto po [18]

programa na različne dele tako, da vsak del rešuje nek ločen problem. Eno komponento predstavlja ena datoteka.

Vsaka programska komponenta je lahko povezana z drugo preko uporabe vmesnikov. V TinyOS ločimo dva tipa komponent: modul in konfiguracijo. Modul je implementacija enega ali več vmesnikov. Vsebuje spremenljivke in izvršljivo kodo. Konfiguracijo pa uporabimo za "sestavljanje" preostalih komponent skupaj, povezovanje vmesnikov, ki jih komponente uporablajo z vmesniki, ki jih ponujajo druge komponente.

Vmesniki so "dvosmerni": vsebujejo listo ukazov, ki jih mora implementirati vmesnik, ki ga ponuja druga komponenta, in listo dogodkov, ki jih mora implementirati komponenta, ki ta vmesnik uporablja. Primer: A -> B pomeni, da ukazi sledijo iz A v B, dogodki pa iz B v A. Na sliki 4.4 je prikazan primer povezave komponent. Vsak program v TinyOS ima glavno konfiguracijo, ki določa, kako so ostale komponente povezane.

Model za sočasno izvajanje omogoča podporo večih komponent istočasno, pri čemer porabi malo pomnilnika. Vsak vhodno / izhodni klic je razdeljen (split-phase). Namesto blokiranja dokler se zahteva ne izvede, se zahteva

vrne takoj in klicatelj dobi povratni poziv, ko se vhodno / izhodni klic izvede. Ker ne čakamo na izvajanje vhodno izhodnih klicev, potrebujemo le en sklad. TinyOS ne podpira večnitnosti. Namesto večnistnosti imamo na voljo opravila (tasks), ki so neke vrste zakasnjeni klici podprogramov. Vsaka komponenta lahko zahteva opravilo, ki ga bo TinyOS izvedel v nekem prihodnjem času.

API nudi splošno funkcionalnost (npr. pošiljanje paketov).

4.5.1 nesC

Ker je nesC (kratica za network embedded systems C) programski jezik za TinyOS, bomo v tem podpoglavju na hitro predstavili njegove posebnosti. Največja razlika med programom, spisanim v programskem jeziku C in nesC je v strukturi: poimenovanju in razporeditvi elementov programa, kot so funkcije in spremenljivke. Koda spisana v jeziku C je sestavljena iz funkcij in spremenljivk v datotekah, ki se prevedejo posebej in nato povežejo. nesC programi so sestavljeni iz komponent, ki jih prevajalnik poveže in prevede kot celoto.

Za prikaz razlik se bomo oprli na primer iz knjige [4]. Oglejmo si, kako bi izgledala psevdo programska koda v jeziku C za program, ki ob zagonu hipotetičnega elementa BSO z LED diodo le-to prižge. Nato pa element postavi v stanje *sleep*.

```
include "mote.h" //knjiznica, kjer imamo vse, kar potrebujemo za
    delo s strojno opremo

int main()
 {
        mote_init(); // inicializacija elementa
        led0_on(); //klic funkcije za prizig LED diode
        sleep(); //element ponovno postavimo v sleep stanje
 }
```

V nesC pa taka koda obsega več datotek.

```
module PowerupC {
      uses interface Boot; //dva vmesnika Boot in Leds - s tema
         je modul povezan z ostalim sistemom
      uses interface Leds;
}
implementation { // nudi implementacijo, ki klice led0On
      event void Boot.booted () {
         call Leds.led0On (); //ker so LED diode ostevilcene v
            TinyOS podamo tudi stevilo, v tem primeru 0 pri
            klicu ukaza
      }
}
```

Zgornja koda predstavlja modul. S preostalim sistemom se modul povezuje z dvema vmesnikoma Boot in Leds. Vsebuje implementacijo za dogodek iz Boot vmesnika in kliče ukaz iz Leds vmesnika. Če primerjamo kodo modula z implementacijo v C-ju vidimo, da je implementacija dogodka *event void Boot.booted()* na mestu glavne funkcije *int main()*. Klic *call Leds.led0On()*; pa ustreza klicu *led0_on()*. Program v C jeziku je sestavljen iz funkcij, nesC pa iz komponent, ki implementirajo določeno funkcionalnost. V podanem primeru je to prižiganje LED diode ob zagonu elementa.

Preprosta vmesnika Boot in Leds sta prikazana v nadaljevanju.

```
interface Boot { //uporabljamo Boot in Leds vmesnika
event void booted(); //z oznako event (dogodek) ali commmand (
    ukaz) locimo med zahtevami in povratnimi klici
}
interface Leds {
command void led0On();
command void led0Off();
command void led0Toggle();
...
}
```

Drugi del programa Powerup predstavlja konfiguracija PowerupAppC, ki določa, kako je modul povezan s TinyOS.

```
configuration PowerupAppC { }
implementation
{
components MainC, LedsC, PowerupC;
MainC.Boot -> PowerupC.Boot;
PowerupC.Leds -> LedsC.Leds;
}
```

Iz konfiguracije je razvidno, da je program zgrajen iz treh komponent: MainC (za sistemski zagon ali angl. "boot"), LedsC (nadzor nad LED diodami) in PowerupC (modul, katerega koda je zapisana zgoraj). V konfiguraciji podamo točne povezave med posameznimi vmesniki, ki jih komponente bodisi ponujajo bodisi uporabljajo. nesC uporablja oznako -> za povezavo med vmesniki. A->B pomeni A se povezuje z B. A je "uporabnik" vmesnika, B pa "ponudnik". *MainC.Boot -> PowerupC.Boot* lahko preberemo kot: vmesnik Boot komponente MainC se povezuje z vmesnikom Boot komponente PowerupC.

Ko se izvede zagon sistema z MainC, sporoči to Boot vmesniku, ki je povezan s PowerupC booted dogodkom. Ta dogodek nato kliče ukaz led0On

v Leds vmesniku, ki je prav tako povezan s konfiguracijo z Leds vmesnikom, ki ga nudi LedsC.

V kodi C kličemo funkcijo *led0_on()*, ki je povezana z določeno knjižnico – če imata to funkcijo dve knjižnici, se izbere eno izmed njih. V konfiguraciji z nesC pa točno določimo katero implementacijo funkcije želimo.

Kodo v programskem jeziku nesC s končnico *.nc* prejavalnik nesC najprej prevede v .c datoteko, nato pa z domorodnim (angl. native) C prevajalnikom v ustrezno obliko, ki jo lahko zaženemo na strojni opremi.

4.5.2 Instalacija TinyOS in vzpostavitev delovnega okolja za delo s TinyOS

V tem podpoglavju se bomo omejili le na osnovno namestitev orodij in predstavili nekaj osnovnih napotkov za delo s TinyOS. Delovno okolje smo vzpostavili pod 64-bitnim operacijskim sistemom Debian. Navodila tako veljajo tudi za operacijski sistem Ubuntu.

Za uporabnike operacijskega sistema Debian je instalacija dokaj preprosta, saj so v repozitorijih na voljo tako imenovani .deb paketi[5].

Vendar velja opozoriti, da so navodila na uradni spletni strani TinyOS Univerze v Stanfordu [13] nekoliko nepopolna za strojno opremo Zolertia Z1. Bolj ustreznna so navodila na spletni strani podjetja Zolertia, s katerimi nisem imela večjih težav [7].

V datoteko sources.list kot superuser (su) dodamo ustrezen repozitorij in z uporabo apt (Advanced Packaging Tool) instaliramo zadnjo verzijo TinyOS (v času pisanja diplomskega dela TinyOS 2.1.2.).

```
$ sudo echo "deb http://tinyos.stanford.edu/tinyos/dists/ubuntu
   lucid main" >> /etc/apt/sources.list
$ sudo apt-get update
$ sudo apt-get install tinyos-2.1.2
```

[5] več o .deb paketih na http://www.debian.org/doc/manuals/debian-faq/ch-pkg_basics.en.html

Preden začnemo z delom, moramo pripraviti spremenljivke okolja (enviornment variables), ki jih bomo potrebovali za izvajanje kode. To naredimo z zagonom skripte tinyos.sh, ki je dostopna na spletni strani Zolertia Wiki [7].

```
$ ./tinyos.sh
Setting up for TinyOS
```

V primeru kasnejših težav z okoljem skripto zaženemo z ukazom: source tinyos.sh. Zaganjanje bash skripte z ukazom source pomeni, da nismo ustvarili nove lupine (shell), kar storimo z ukazom ./. Spremenljivke okolja se zato spremenijo v trenutni lupini.

Z ukazom motelist preverimo, ali smo uspešno priključili tudi strojno opremo – priklopljenih imamo lahko več senzorjev istočasno.

```
$ motelist
Reference  Device           Description
---------- ---------------- ----------------------------
Z1RC1048   /dev/ttyUSB0     Silicon Labs Zolertia Z1
Z1RC1107   /dev/ttyUSB1     Silicon Labs Zolertia Z1
```

Sedaj lahko začnemo s pisanjem, prevajanjem kode in nalaganjem operacijskega sistema in programske kode na element Z1. Pri tem moramo paziti le še na dovoljenja (permissions) in v primeru, da je to potrebno, dovoljenja uredimo z uporabo ukaza chmod.

```
serial.serialutil.SerialException: could not open port /dev/
    ttyUSB0: [Errno 13] Permission denied: '/dev/ttyUSB0'
make: *** [program] Error 1
$ su
Password:
# chmod a+rw /dev/ttyUSB0
# exit
```

Za TinyOS razvijamo kodo s programskim jezikom nesC, ki smo ga že predstavili v podpoglavju 4.5.1.

V direktoriju /apps/tests/z1 imamo več uporabnih primerov programske kode. Natančneje poglejmo test temperaturnega senzorja TMP102, ki smo ga podrobneje predstavili v podpoglavju 3.3. V mapi /Temperature imamo tako na voljo TestTmp102C.nc, ki ga prevedemo z ukazom *make <platforma>*. Niz <platforma> ustrezno nadomestimo z oznako izbranega elementa. Nalaganje TinyOS in prevajanje programske kode lahko opravimo s kombinacijo ukazov *make <platforma> install*. Na voljo imamo tudi ukaz *make <platforma> reinstall*, s katerim na modul naložimo nazadnje prevedeno kodo. Z ukazom *make clean* lahko počistimo direktorij vseh že generiranih datotek (v tem primeru moramo za reinstall ponovno pognati *make <platfrorma>*).

```
$ make z1 install
```

V primeru, ki smo ga izbrali za zgled imamo z ukazom *printfUART* realiziran izpis. Ukaz printfUART je ukaz *printf* za UART (universal asynchronous receiver/transmitter) in zanj velja podobno kot za ukaz *prinf* z nekaj sremembami. Vključiti ga moramo z *include printfUART.h* in inicializirati s *printfUART_init()*. Paziti moramo, da tudi v enovrstičnih pogojnih stavkih oziroma zankah uporabimo oklpaje, primer:

```
if (x < 3)  {printfUART("Vrednost x: %i\n", x);}
```

in v kolikor izpisujemo le tekst, moramo dodati še dodatna narekovaja

```
printfUART("Test \n", "");
```

Za Zolertio Z1 je na voljo tudi ukaz *printfz1*, kjer zgornja omejitev ne velja. Prav tako pa ga moramo inicializirati v *event void Boot.booted()* s

printfz1_init() in seveda vključiti *include printfZ1.h*. V obeh primerih ne pozabimo v datoteki Makefile vključiti vrstice
CFLAGS += -DPRINTFUART_ENABLED.

S tem ukazom dobimo možnost razhroščevanja in spremljanja delovanja senzorja, ki je priklopljen preko USB. Najlažje "poslušamo" dogajanje s pomočjo programov kot sta PuTTY ali picocom. Ukazu določimo serijsko hitrost podatkov in napravo ter lahko poslušamo dogajanje na serijskem vhodu. V kolikor nismo sigurni kakšna je baudna hitrost za našo napravo, lahko to preverimo z ukazom stty.

```
$ stty -F /dev/ttyUSB1
speed 115200 baud; line = 0;
```

Po uspešnem prevajanju in nalaganju zgleda TestTmp102C, se na element Z1 lahko povežemo in spremljamo meritve temperature.

```
$ picocom -b 115200 /dev/ttyUSB0
picocom v1.7
port is        : /dev/ttyUSB0
flowcontrol    : none
baudrate is    : 115200
parity is      : none
databits are   : 8
escape is      : C-a
local echo is  : no
noinit is      : no
noreset is     : no
nolock is      : no
send_cmd is    : sz -vv
receive_cmd is : rz -vv
imap is        :
omap is        :
emap is        : crcrlf,delbs,
Terminal ready
Temp: 27.22
```

Branje temperature s privzetim primerom

Program, zapisan v nesC jeziku je zgrajen iz dveh različnih komponent: modula in konfiguracije, kot smo predstavili v podpoglavju 4.5.1. Modul je implementacija enega ali več vmesnikov, konfiguracija pa služi povezovanju ostalih komponent, povezuje vmesnike, ki jih uporablja komponenta z drugimi vmesniki drugih komponent. Vmesnik, ki ustreza določeni komponenti, predstavljaja funkcionalnosti, ki jih le-ta potrebuje. Določa ime komponente, ukaze in dogodke, ki jih bo komponenta uporabljala.

V mapi /Temperature, kjer se nahaja zgoraj opisani primer vidimo tako dve datoteki (poleg makefile in README): TestTmp102C.nc (modul) in TestTmp102AppC.nc (konfiguracija). Vsi programi potrebujejo glavno konfiguracijsko datoteko, ki se praviloma imenuje po samem programu in jo

prevajalnik nesC uporabi pri prevajanju. Čeprav bi za imena lahko uporabili karkoli, se priporoča, da se zaradi preglednosti drži tega nenapisanega pravila. Konfiguracija poveže implementacijo z ostalimi komponentami, ki jih potrebujemo. Modul pa vsebuje dejansko implementacijo programa.

Delitev na module in konfiguracije omogoča programerju, da uporabi že obstoječe implementacije, tako bi v teoriji lahko zgradili program le z povezovanjem večih modulov, ki pa jih nismo sami razvili. V primeru Temperature pa imamo konfiguracijo in modul, ki sta "par".

Koda konfiguracije TestTmp102AppC.nc je prikazana spodaj.

```
configuration TestTmp102AppC {} //ime konfiguracije
implementation { //implementacija konfiguracije
  components MainC, TestTmp102C as App, LedsC; //komponente, ki
     jih konfiguracija oznacuje (reference)
  App.Leds -> LedsC;  //povezava med vmesniki, ki jih TestTmp102
      komponenta uporablja z vmesniki, ki jih nudi komponenta
     LedsC

  App.Boot -> MainC.Boot;
  components new TimerMilliC() as TestTimer; //casovna
     komponenta
  App.TestTimer -> TestTimer;

  components new SimpleTMP102C() as Temperature;
  App.TempSensor -> Temperature;
}
}
```

Oglejmo si še začetni del programske kode modula TestTmp102C.nc.

```
module TestTmp102C { //ime modula
  uses { //katere vmesnike bomo uporabili - kateregakoli od
     nastetih lahko uporabimo
    interface Leds;
    interface Boot;
    interface Timer<TMilli> as TestTimer;
    interface Read<uint16_t> as TempSensor;
  }
}
implementation {
  // implementacija
}
```

S primerom "UDPTemp", ki je opisan na spletni strani [7] bi lahko implementirali merjenje temperature kot bomo to storili v pričujočem delu v nadaljevanju s Contiki operacijskim sistemom. Višji nivo v nadaljevanju opisane sistemske rešitve je neodvisen od realizacije na nižjem nivoju.

4.6 Contiki

Contiki je odprtokoden operacijski sistem. Namenjen je sistemom, ki imajo omejene vire, osredotočen pa je predvsem na energijsko učinkovite naprave za tako imenovan "Internet of Things"[6].

Contiki ima na voljo dva omrežna sklada: uIP (in UIPv6) in Rime [19]. Njegova posebnost je podpora protokolnega sklada TCP/IP (uIP ali micro IP, version 4 in 6), s katero omogoča povezavo preko interneta in protokol IPv4 (Internet Protocol version 4) ali IPv6. Seveda ne omogoča vseh funkcionalnosti TCP/IP modela, predvsem ne tistih, ki potrebujejo več pomnilnika (kot je na primer IP fragmentacija). Rime je "lahki" omrežni sklad za na-

[6]"Internet stvari" ali angl. *Internet of Things* je izraz, ki opisuje medsebojno povezanost naprav oziroma objektov v omrežni strukturi podobni internetu ali povezanih na internet. Prvi je predlagal besedno zvezo Kevin Ashton leta 1999, koncept pa je znan vse od leta 1991 [5]

prave z omejenimi viri, ki podpira načine komunikacije tipične za BSO (single hop unicast, broadcast, network flooding,ipdr. Več o protokolih, primernih za BSO v knjigi [1]). V primeru multihop komunikacije omogoča aplikaciji, da uporabi svoje protokole, ki sicer niso vključeni v Rime sklad. Rime uporabimo v primerih, ko je TCP/IP preveč požrešen ali ga enostavno ne potrebujemo.

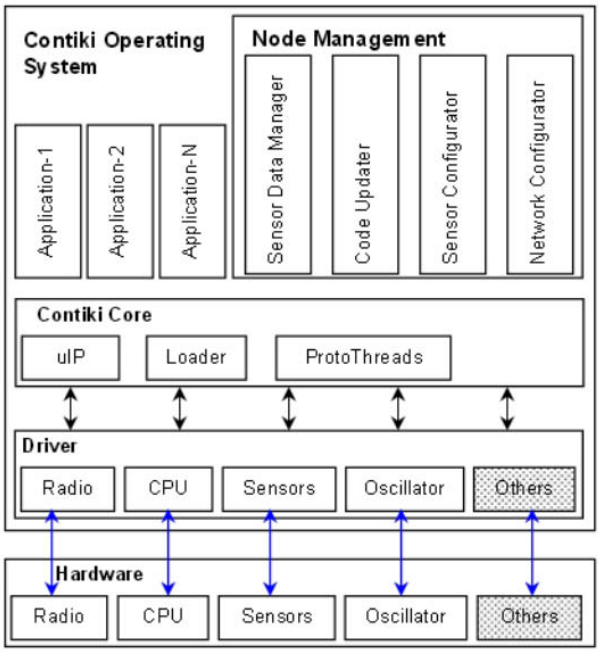
Pomembna lastnost operacijskega sistema Contiki je možnost večnitenja nad jedrom (angl. kernel) sistema. Z implementacijo protothreadov nam Contiki omogoči sekvenčno programiranje in linearno izvajanje, kar smo že spoznali v poglavju 4.4.2. Jedro operacijskega sistema vsebuje časovni razporejevalnik dogodkov (angl. event scheduler), ki razporeja dogodke procesom in periodično kliče opravilnike. Vsa koda, ki se izvrši, se izvrši bodisi zaradi klica jedra ali dela opravilnika. Jedro nikoli ne prekine opravilnika dogodka.

Za razliko od TinyOS, ki je zasnovan monolitsko, je Contiki modularen (slika 4.5) [11]. Koncept modularnega operacijskega sistema je bližje pojmovanju klasičnega operacijskega sistema, saj to pomeni, da se programska koda prevede individualno in jo lahko jedro operacijskega sistema naloži.

4.6.1 Instalacija Contiki OS in vzpostavitev delovnega okolja

Najlažja in najbolj zanesljiva rešitev za delo z operacijskim sistemom Contiki je uporaba "*Instant Contiki*" paketa. Gre za Vmware virtualni stroj, ki poganja Ubuntu Linux in ima nameščen Contiki OS ter vsa razvojna orodja, ki jih potrebujemo. Za začetek sledimo navodilom na uradni spletni strani [19]. Najprej naložimo VMWare player, s katerim bomo zagnali virtualni stroj, ki ga najdemo na [20] - izberemo najnovejšo različico. V kolikor uporabljamo eno izmed distribucij Linuxa, svetujemo uporabo programa VirtualBox. Z uporabo VMWare playerja sem sama imela namreč težave pri priklopu elementa Zolertia Z1 v virtualni stroj.

Vsebino datoteke *Instant Contiki*, ki je formata .zip, odpakiramo na poljubno lokacijo na trdem disku. Z Vmware player programom odpremo da-



Slika 4.5: Poenostavljen shematični prikaz arhitekture OS Contiki, vir [17]

toteko .vmx. Geslo, ki ga potrebujemo za vstop je "user". Delovno okolje si prilagodimo po želji.

Za začetek dela z elementom Zolertia Z1 moramo najprej virtualnemu stroju omogočiti dostop do naprave. V direktoriju /contiki-2.7/examples imamo na voljo primere programske kode. Najlažje preverimo ali naša povezava z elementom preko USB priključka deluje z enostavnim primerom iz mape /examples. Vzemimo primer poimenovan "hello-world". Program hello-world.c ob zagonu elementa izpiše besedno zvezo "hello world" na stdout (standard output ali standardni izhod).

V mapi primera (/contiki-2.7/examples/hello-world) vidimo štiri različne datoteke:

- hello-world.c (koda, ki jo bomo uporabili)
- hello-world-example.csc (datoteka za Cooja simulacijo)
- Makefile
- README.md

Contiki "build system" je sestavljen iz večih Makefile datotek[7]. Te datoteke so:

- **Makefile**: makefile našega projekta, ki mora biti v direktoriju, kjer se nahaja koda
- **Makefile.include**: sistemski Contiki makefile, ki se mora nahajati v glavnem imeniku (root) drevesa izvorne kode
- **Makefile.$(TARGET)**, kjer je $(TARGET) ime elementa, za katerega prevajamo kodo: vsebuje pravila, ki so specifična za določen element (na primer Zolertia Z1) in se mora nahajati v poddirektoriju elementa (/contiki-2.7/platform/z1/Makefile.z1). Makefile.z1 vsebuje kar nekaj uporabnih ukazov, med drugim:
  - *z1-motelist*: Seznam vseh naprav Z1, ki so priključene na računalnik
  - *z1-reset*: Ponovni zagon vseh Z1 naprav priklopljenih na računalnik
  - *login*: Izpis serijskega izhoda
  - *serialdump*: Ukaz kot login, le da pred vsako vrstico izpisa doda čas izpisa (Unix timestamp).
- **Makefile.$(CPU)** kjer je $(CPU) ime družine mikrokontrolerja, ki ga uporablja element, za katerega razvijamo kodo: nahajati se mora v poddirektoriju /cpu (/contiki-2.7/cpu/msp430/Makefile.msp430)
- **Makefile.$(APP)** kjer je $(APP) ime aplikacije, ki je v /apps direktoriju.

Makefile primera "hello-world" je preprost. Definiramo CONTIKI – lokacijo drevesa izvorne kode, ime našega programa in na koncu vključimo še sistemski Makefile.include.

```
CONTIKI_PROJECT = hello-world
all: $(CONTIKI_PROJECT)
CONTIKI = ../..
include $(CONTIKI)/Makefile.include
```

[7]Makefile je datoteka, ki vsebuje pravila, ki jih ukaz make potrebuje, da avtomatično zgradimo (build) / prevedemo program v izvršljiv program

Makefile datoteke nam omogočajo, da lahko prevedemo kodo za različne elemente in z različnimi parametri.

Na podlagi hello-world primera si oglejmo še, kako deluje Contiki *make*. Z ukazom make celotno kodo prevedemo v delujoč operacijski sistem, ki vsebuje našo kodo. Če ne uporabimo dodatnih parametrov, bomo uporabili "native target", ki je Contiki, kot je razvidno iz kode spodaj. Ta nam omogoča, da prevedemo kodo kot program, ki ga bomo lahko pognali znotraj operacijskega sistema, kjer kodo razvijamo.

```
user@instant-contiki:~/contiki-2.7/examples/hello-world$ make
TARGET not defined, using target 'native'

user@instant-contiki:~/contiki-2.7/examples/hello-world$ ./hello
    -world.native
Contiki 2.7 started
Rime started with address 2.1
MAC nullmac RDC nullrdc NETWORK Rime
Hello, world
```

Če želimo kodo prevesti in naložiti na določen element, ki poganja Contiki operacijski sistem, moramo določiti za kateri element gre, z ukazom *make TARGET=element*. Da bi se izognili tipkanju ukaza vsakič znova, ustvarimo datoteko Makefile.target v direktoriju projekta z ukazom *make TARGET=element savetarget*.

```
user@instant-contiki:~/contiki-2.7/examples/hello-world$ make
    TARGET=z1 savetarget
saving Makefile.target
```

Sedaj moramo kodo prevesti. Kodo prevedemo z ukazom *make hello-world* (dodamo lahko tudi parameter -s za t.i. "silent mode", v kolikor ne želimo podrobnega izpisa na zaslonu), na element Z1 pa ga naložimo z ukazom *make hello-world.upload*. Oboje hkrati lahko storimo z uporabo

zadnjega ukaza. Poskrbeti moramo tudi za ustrezna dovoljenja. V primeru napake:

```
serial.serialutil.SerialException: could not open port /dev/
    ttyUSB0: [Errno 13]
Permission denied: '/dev/ttyUSB0'
```

kot superuser ali root spremenimo ustrezna dovoljenja:

```
root@instant-contiki:/home/user/contiki-2.7/examples/hello-world
    # chmod a+rw /dev/ttyUSB0
```

in ponovno poženemo ukaz *make hello-world.upload*. Sedaj nas zanima še, kje lahko vidimo izpis, ki ga Z1 pošilja na standardni izhod. Z ukazom make login odpremo serijska vrata, kjer imamo priklopljen element. Ob ponovnem zagonu elementa lahko s pomočjo ukaza *make login* vidimo izpis, kot je prikazano spodaj.

```
user@instant-contiki:~/contiki-2.7/examples/hello-world$ make
    login
using saved target 'z1'
../../tools/sky/serialdump-linux -b115200 /dev/ttyUSB0
connecting to /dev/ttyUSB0 (115200) [OK]
Rime started with address 0.0
MAC 00:00:00:00:00:00:00:00 Contiki 2.7 started. Node id is not
    set.
CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
Starting 'Hello world process'
Hello, world
```

Zgradba programa Contiki OS je na prvi pogled nekoliko preprostejša od komponent TinyOS. Vzemimo omenjeni primer "hello-world" za začetni zgled. Spodnji zgled smo ustrezno opremili s komentarji.

```
#include "contiki.h" // contiki.h moramo vkljuciti vedno!

#include <stdio.h>  //knjiznica za ukaz printf()

PROCESS(hello_world_process, "Hello world process"); //
    definicija procesa, prvi argument je ime procesa, drugi pa
    referenca, string za razhroscevanje
AUTOSTART_PROCESSES(&hello_world_process); //proces avtomaticno
    zazenemo vsakic, ko se Contiki zazene

PROCESS_THREAD(hello_world_process, ev, data) //procesna nit,
    vkljuciti moramo ime procesa in spremenljivke dogodkov - ev
    vsebuje stevilko dogodka, v data pa je ustrezen kazalec (
    pointer)
{
  PROCESS_BEGIN(); //zacetek procesa, pred PROCESS_BEGIN v
      vecini primerov, ne potrebujemo kode, ce bi jo vkljucili,
      bi se zagnala vsakic, ko bi zagnali proces

  printf("Hello, world\n"); //z ukazom printf na standardni
      izhod izpisemo niz

  PROCESS_END(); //konec procesa, proces moramo vedno zakljuciti
       s tem ukazom, saj se sicer koda ne bo prevedla
}
```

Zapisana koda nam izpiše želeni niz le ob zagonu programa, ker se celotna nit izvede le enkrat. V splošnem večkrat srečamo primere, kjer se izvajanje niti ponavlja, ali pa se celo nikoli ne konča (velikokrat uporabimo neskončno zanko), razen če celoten sistem ugasnemo ali proces ubije drug proces s klicem *process_exit()*.

Oglejmo si, kako bi kodo iz primera spremenili, da bi neprekinjeno izpisovali želeni niz enkrat na določen časovni interval.

Uporabimo funkcijo event timer (časovnik dogodka): *etimer_set (struct etimer * et, clock_time_t interval)*. Parametra sta et - kazalec na časovnik

dogodka in interval - čas, preden časovnik poteče. Definiramo etimer et, za začetek procesa dodamo neskončno zanko in nastavimo časovnik dogodka. S klicem funkcije *PROCESS_WAIT_EVENT_UNTIL* čakamo na dogodek. Koda spodnjega primera bo izpisovala niz vsakih deset sekund.

Knjižnica ure (clock library) v Contiki OS skrbi za prevajanje sekund v takt ustreznega mikrokontrolerja izbranega elementa. CLOCK_SECOND je ena sekunda, izmerjena v sistemskem času mikrokontrolerja elementa.

```
while(1){
    static struct etimer et;
    etimer_set(&et, CLOCK_SECOND * 10);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    printf("Hello, world\n");
}
```

Z uporabo ukaza *make serialdump* lahko izpisu dodamo Unix timestamp, ki nam prikaže število sekund, ki so pretekle od 1. 1. 1970. Če želimo časovni izpis našega lokalnega časa, lahko ustrezno spremenimo datoteko timestamp, ki se nahaja v */contiki-2.7/tools*. Gre za preprosto perl skripto, kjer time zamenjamo z localtime:

```
#!/bin/sh
# We run perl through a shell to avoid having to hard-code the
   path to perl
perl -e '$|=1; while(<>) {print localtime . " $_";}'
```

Ukaz serialdump nam izpis tudi shranjuje v datoteko z imenom *serialdump-‘date +%Y%m%d-%H%M‘* v trenutni direktorij projekta. V kolikor želimo ime datoteke ali lokacijo spremeniti, si moramo podrobneje ogledati Makefile.common, ki se nahaja v */contiki-2.7/platform/z1* mapi. Osnovna sintaksa Makefile-a je sledeča:

```
target: dependencies
[tab] system command
```

Naš "target" je serialdump, ukaz (system command) pa mu sledi v naslednji vrstici, začenši s tabulatorjem. Vidimo, da uporabi za pisanje v datoteko ukaz tee, kateremu lahko dodamo novo lokacijo datoteke.

```
$(SERIALDUMP) -b115200 $(USBDEVPREFIX)$(word $(MOTE), $(CMOTES))
| $(CONTIKI)/tools/timestamp
| tee /home/user/Desktop/serialdump-`date +%Y%m%d-%H%M`
```

Sedaj imamo osnovno ogrodje za izdelavo dela sistema omrežja za merjenje temperature. V nadaljevanju bomo prototip merjenja temperature z BSO predstavili podrobneje.

4.7 Primerjava TinyOS in Contiki

V tem podpoglavju bomo na kratko predstavili nekaj glavnih razlik med obema operacijskima sistemoma. Navedla bom tudi nekaj svojih osebnih izkušenj in mnenj.

Oba operacijska sistema lahko tečeta na mikrokontrolerjih z zelo omejenimi viri, vendar je TinyOS bolj primeren za sisteme z najbolj omejenimi viri in kjer je majhna poraba pomnilnika in energije izredno ključnega pomena. Contiki pa je lahko boljša izbira v primeru, da želimo večjo fleksibilnost, kot na primer redno posodabljanje programske opreme na večih elementih omrežja [16].

TinyOS sloni na arhitekturi, pogojeni dogodkom in s tem omogoča sočasnost, Contiki pa poleg tega omogoča tudi pomnilniško varčno večnitnost nad jedrom, ki je pogojen dogodkom. TinyOS s knjižnico TOSThreads [14] sicer omogoča večnitnost, a z večjo porabo pomnilnika.

Če želimo nadgraditi že naloženo programsko kodo na module v BSO,

lahko to v Contikiju storimo dinamično in naložimo le spremenjeni del programa. TinyOS pa moramo ponovno namestiti v celoti, ker so komponente povezane s celotnim sistemom.

Ker je za razvijalca programski jezik zelo pomemben, velja omeniti tudi razliko med jezikoma nesC in C. Najbolj očitna je strukturna razlika. Pri programskem jeziku C imamo spremenljivke, funkcije, tipe definirane v datotekah, ki se prevedejo vsaka zase in nato povežejo. V nesC pa so programi sestavljeni iz komponent (glej poglavje 4.5.1). nesC prevajalnik jih združi, nato jih prevajalnik C prevede kot eno samo datoteko. Za programiranje v Contiki OS nam dobro znanje jezika C in poznavanje dela s kazalci ali pointerji lahko zelo koristi, v nasprotnem primeru pa lahko zabredemo v težave in je nesC bolj smiselna izbira.

Vzpostavitev delovnega okolja za oba operacijska sistema po mojih izkušnjah ni potekala povsem brez težav. Zelo hitro se zgodi, da izgubimo veliko časa zaradi malenkosti, ki se za delovno okolje izkažejo pomembne, vendar niso nikjer podrobneje opisane. V veliki meri sem si pomagala s spletno stranjo Zolertia Wiki [7]. Pomembno je, da se pred izbiro operacijskega sistema ali strojne opreme ustrezno pozanimamo o dostopnosti in kvaliteti informacij, ki so nam na voljo.

TinyOS ima po mojem mnenju boljšo podporo v smislu informativnih spletnih strani, ki nudijo odgovore na pogosta vprašanja. Pomembni so tudi preprosti primeri, ki jih lahko uporabimo pri nadaljnem delu. Iskalnik Google avgusta 2014 vrne približno 749,000 zadetkov za iskalni niz "TinyOS", medtem ko jih za "Contki" preštejemo približno 435,000.

Pri Contiki operacijskem sistemu smo po mojem mnenju morda malenkost bolj prepuščeni sami sebi. Sklepam sicer, da se bo podpora uporabnikom za Contiki utegnila spremeniti na bolje, saj so razvijalci odprli podjetje "Thingsquare". Za tržne uspehe bo potrebno uporabniku približati in izboljšati tudi Contiki podporo.

Oba operacijska sistema sta na voljo tudi na priljubljenem spletnem mestu GitHub, kjer je vidna večja aktivnost na projektu TinyOS.

Za spoznavanje načina dela z brezžičnimi senzorskimi omrežji in dela na sistemih z omejenimi viri sta po mojem mnejnju primerna oba operacijska sistema. Pri obeh, tako pri TinyOS kot Contiki, imamo za določeno strojno opremo že na voljo gonilnike, kar pomeni, da lahko začnemo z delom brez podrobnejšega predznanja o strojni opremi.

TinyOS bi priporočili v primeru, da imamo na voljo res malo virov in nam je pomembno ohraniti vsak pomnilniški bit. Contiki pa bi izbrali zaradi fleksibilnosti in predvsem v primeru, ko bi zaradi narave problema morali spreminjati programsko kodo vozlišč BSO. Če želimo sistem, ki bo bolje skrbel za porabo električne energije ima TinyOS veliko prednost. V Contikiju moramo za mehanizme ohranjanja enegije večinoma poskrbeti sami.

4.8 Drugi operacijski sistemi za BSO

V tem podpoglavju bomo našteli in na kratko opisali nekatere izmed operacijskih sistemov za BSO. Izbrali smo tiste, ki se pogosteje pojavijo v strokovni literaturi. V tabeli 4.2 primerjamo pet od predstavljenih OS [11], vključno z operacijskima sistemoma TinyOS in Contiki.

4.8.1 MANTIS

MANTIS (*http://mantisos.org/*) je okrajšava za MultimodAl system for NeTworks of In-situ wireless Sensors. MANTIS je lahek in energijsko učinkovit sistem, ki skupaj z jedrom, časovnikom in omrežnim protokolnim skladom potrebuje 500 bajtov RAM pomnilnika. Energijsko učinkovitost pa doseže s klicanjem MOS sleep() funkcije, ko se izvedejo vse niti.

4.8.2 LiteOS

LiteOS (*http://www.liteos.net/*) je operacijski sistem za delo v realnem času (real-time operating system ali krajše RTOS). Gre za UNIX-like sistem, ki ga lahko namestimo na sisteme z omejenimi viri. Prednost sistema je prepo-

znavnost Unix sistemov, saj je upravljanje z brezžičnim senzorskim omrežjem v primeru uporabe LiteOS podobno programiranju v okolju Unix (in programskem jeziku C). Sestavljen je iz treh komponent: LiteShell (Unix-like lupina, ki nudi podporo za lupinske ukaze), LiteFS (datotečni sistem) in jedra. Posebnost LiteOS je, da se LiteShell nahaja na osebnem računalniku (ali glavnem elementu omrežja) in ga lahko uporabljamo, če smo kot uporabnik prijavljeni na tem osebnem računalniku. Dostopen je na Google Code: *http://code.google.com/p/liteos/*.

4.8.3 Nano-RK

Cilji pri izdelavi Nano-RK (RK = Resource Kernel) so bili večopravilnost, podpora multi-hop omrežjem in majhna velikost sistema. Nano-RK zasede 2 Kb RAM pomnilnika in 18 Kb ROM pomnilnika. Napisan je v programskem jeziku C. RK določa, koliko časa se lahko določen sistemski vir uporablja, tako se lahko nek program izvaja le npr. 10 ms ali pošiljanje sporoči obsega le določeno število paketov v določenem časovnem okvirju. Več informacij, med drugim tudi navodila za namestitev okolja in osnovna navodila za Nano-RK se nahajajo na *http://www.nano-rk.org*.

4.8.4 eCos

Ime eCos je kratica za "embedded configurable operating system". Gre za prilagodljiv operacijski sistem, ki ga lahko predelamo v smislu zahtev našega problema. V začetku ga je razvijalo podjetje Red Hat, zaradi česdar obstaja zmotno prepričanje, da je eCos osnovan na Linux sistemu. Razvijalci so ustanovili tudi podjetje in komercialno različico sistema, eCosPro. Vse kar potrebujemo za začetek dela z eCos najdemo na spletni strani *http://ecos.sourceware.org/getstart.html*.

OS	Arhitektura	Programiranje	Čas. Razvrščanje	Komunikacija	Simulator	Prog. jezik
TinyOS	Monolitska	pogojeno z dogodkom	FIFO	Active message	TOSSIM	nesC
Contiki	Modularna	protothreads, dogodki	ko se dogodek zgodi, v primeru prekinitve prioriteta	uIP, Rime	Cooja	C
MANTIS	Večplastna	Z nitmi	Glede na prioriteto 5 razredov	COMM (kernel), networking layer	preko AVRORA	C
Nano-RK	Monolitska	Z nitmi	RMS (Rate-monotonic) in rate harmonized	Socket like abstraction	/	C
LiteOS	Modularna	Z nitmi, dogodki	Round Robin (s prioriteto)	File based	preko AVRORA	LiteC++

Tabela 4.2: Primerjava nekaterih operacijskih sistemov za BSO, povzeto po [11]

Poglavje 5

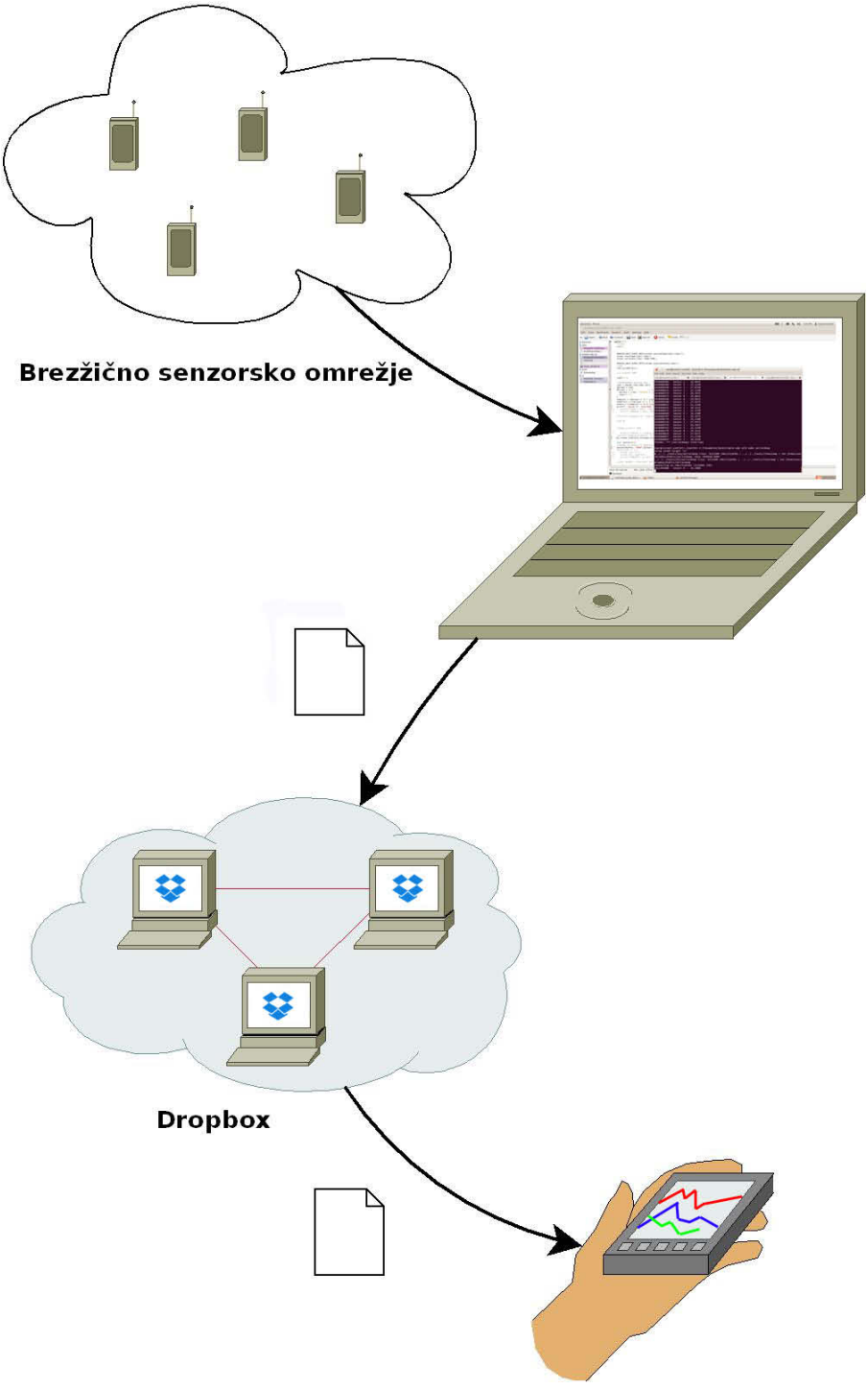
Izdelava prototipa BSO

V praktičnem delu diplomskega dela bomo predstavili izdelavo prototipa BSO. Naš cilj je delujoče BSO, ki ga je mogoče nadzorovati z mobilno napravo.

Omrežje smo realizirali s štirimi elementi Zolertia Z1 z nameščenim operacijskim sistemom Contiki. Za nadzor in spremljanje dogodkov v omrežju smo uporabili mobilni telefon z nameščenim operacijskim sistemom Android. Za povezavo med mobilnim telefonom in ponorom BSO pa smo se oprli na storitev podjetja Dropbox. Na sliki 5.1 je prikazana shema prototipa omrežja. Na sliki vidimo, da s pomočjo računalnika shranjujemo prejete podatke iz BSO v datoteko, ki je dostopna preko storitve Dropbox. Z aplikacijo na mobilnem telefonu datoteko odpremo in podatke ustrezno prikažemo.

Prvi korak pri izdelavi protipa predstavlja vprašanje, kakšen je namen omrežja. Ker smo imeli na voljo elemente Zolertia Z1, smo to izbiro prilagodili strojni opremi in se odločili za preprost problem merjenja temperature v različnih prostorih stanovanja.

Naslednji korak je izbira ustrezne programske opreme oziroma operacijskega sistema za BSO. Ker sem se z operacijskim sistemom TinyOS že srečala v okviru vaj pri predmetu Brezžična senzorska omrežja, sem se odločila za operacijski sistem Contiki. Oba operacijska sistema sta predstavljena v poglavju 4.



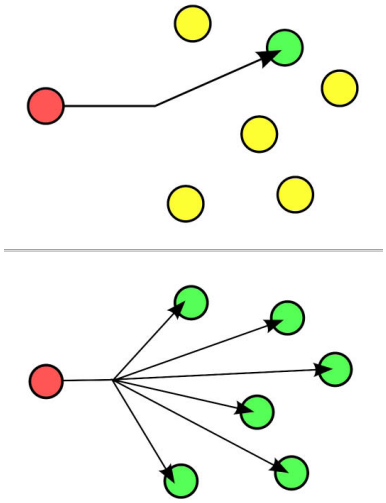
Slika 5.1: Shematski prikaz prototipa

Ko imamo izbran operacijski sistem, elemente in vemo, kaj od omrežja zahtevamo, sledi izbira načina komunikacije med elementi. Ker smo se omejili le na izdelavo prototipa, smo izbirali med že implementiranimi in enostavnimi protokoli za Contiki. Odločili smo se za UDP protokol, ki je za BSO bolj primeren kot TCP. UDP (User Datagram Protocol) je protokol za prenos paketov, kjer strežnik pošlje paket odjemalcu, ne da bi preverjal, ali so bili prejšnji paketi uspešno prejeti. UDP je uporaben v primerih, ko ne potrebujemo popravljanja napak in kjer je pomembno, da ne čakamo na izgubljene pakete. S tem poenostavimo implementacijo operacijskega sistema. Nekaj osnovnih podatkov o UDP je dostopnih na [5], o protokolih primernih za BSO si lahko bralec več prebere v [1]. Ker je za naše potrebe merjenja temperature v stanovanju povsem nepotrebna korekcija napak, prav tako izguba paketov ni pomembna, UDP zadosti našim zahtevam.

Za izdelavo prototipa BSO uporabimo le štiri elemente Zolertia Z1. Zato lahko pakete pošiljamo z vsakega elementa vsem elementom, ne da bi nas moralo skrbeti poplavljanje omrežja. Izbira *broadcast* načina je za nas tudi bolj smiselna, saj si želimo čimbolj enostavne rešitve. Katerikoli Zolertia Z1 element bomo priključili na računalnik, bo sprejemal pakete z izmerjenimi temperaturami ostalih elementov v omrežju. Elementi se tako med seboj ne razlikujejo. To pa je povzročilo manjšo težavo pri zapisovanju izmerjene temperature. Da bo problem merjenja temperature ustrezno rešen, je potrebno natančno vedeti meritev katerega elementa smo zabeležili. To smo rešili z enolično določenimi ID številkami v sami programski kodi. Na vsak element naložimo malenkost drugačno kodo. Uporabljena rešitev je za izvedbo prototipa dovolj dobra in najenostavnejša, vendar nikakor ne priporočljiva ali v splošnem uporabna.

Vzpostavitev delovnega okolja in pričetek dela s Contiki operacijskim sistemom smo predstavili že v podpoglavju 4.6.1. V mapi simple-udp-rpl[1], ki se nahaja na /contiki-2.7/examples/ipv6, imamo že implementirana primera

[1]Koda je dostopna tudi na spletnem naslovu *https://github.com/contiki-os/contiki/blob/master/examples/ipv6/simple-udp-rpl/broadcast-example.c*



Slika 5.2: Na sliki zgoraj je shematski prikaz *unicast* ali pošiljanje enemu elementu, spodaj pa *broadcast* ali pošiljanje vsem elementom omrežja. Vir [5]

broadcast ter unicast (razlika med načinoma pošiljanja paketov je grafično prikazana na sliki 5.2) z uporabo UDP protokola.

Simple-udp modul, ki je del uIP protokolnega sklada ima tri funkcije. Za registriranje UDP povezave je na voljo funkcija:

```
int simple_udp_register (struct simple_udp_connection *c,
    uint16_t local_port, uip_ipaddr_t *remote_addr, uint16_t
    remote_port, simple_udp_callback receive_callback)
```

kjer so parametri:

- c: kazalec na simple_udp_connection
- local_port: lokalna UDP port vrata
- remote_addr: IP naslov, ta parameter nastavimo na NULL v kolikor žeimo prejemati pakete s kateregakoli naslova
- remote_port: oddaljena UDP vrata
- receive_callback: kazalec na funkcijo za prihajajoče pakete

Za pošiljanje paketa na IP naslov, če je bil ta določen s *simple_udp_register()* in na UDP vrata, določena s *simple_udp_register()* imamo na voljo funkcijo:

```
int simple_udp_send (struct simple_udp_connection *c, const void
    *data, uint16_t datalen)
```

s parametri:

- c: kazalec na simple_udp_connection
- data: kazalec na podatek, ki ga pošiljamo
- datalen: dolžina podatka

Za določen IP naslov je na voljo funkcija z istimi parametri kot zgornja, le da imamo še parameter *to*, IP naslov prejemnika.

```
int simple_udp_sendto (struct simple_udp_connection *c, const
   void *data, uint16_t datalen, const uip_ipaddr_t *to)
```

Kot paket želimo periodično pošiljati temperaturo v °C v določenem intervalu. Branje temperaturnega registra smo podrobneje predstavili v podpoglavju 3.3. Interval pošiljanja pa definiramo z

```
#define SEND_INTERVAL    (60 * CLOCK_SECOND)
#define SEND_TIME        (random_rand() % (SEND_INTERVAL))
```

kjer je CLOCK_SECOND sekunda sistemskega časa (torej dejanska sekunda). V zgornjem primeru 1 minuta. V neskončni zanki periodično pošiljamo pakete.

```
while(1) {
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&periodic_timer));
    etimer_reset(&periodic_timer);
    etimer_set(&send_timer, SEND_TIME);

    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&send_timer));
    uip_create_linklocal_allnodes_mcast(&addr);
    simple_udp_sendto(&broadcast_connection, "DATA", 4, &addr);
  }
```

Nekaj težav sem imela pri sestavljanju paketa. Na koncu smo se odločili da pošiljam kar niz oziroma string, saj je bilo tako lažje zaradi t.i. null terminatorja (C String). S tem se namreč tudi konča paket in o njegovi dolžini ni dvoma. Vrednosti temperaturnega registra dodamo še zaporedno številko elementa, ki je temperaturo izmeril in s tem dobimo vsebino podatka.

```
//prebrana vrednost temperaturnega registra
raw1 = tmp102_read_temp_raw();
char *buffer[20];
//dodamo ID elementa k vrednosti raw1
sprintf(buffer, "%d%d",id,raw1);
simple_udp_sendto(&broadcast_connection, &buffer, 20, &addr);
```

V callback funkciji, ki jo kličemo za prihajajoče pakete, z ukazom printf ustrezno preračunano vrednost izmerjene temperature in številko elementa, ki jo je poslal, izpišemo.

```
//prejeti podatek
int16_t received = atoi(data+1);
//na zacetku je ID elementa
char senzor_id = data[0];
//preracunamo vrednost temperature za izpis
sign = 1;
absraw = received;
if(received < 0) {
  absraw = (received ^ 0xFFFF) + 1;
  sign = -1;
}
tempint = (absraw >> 8) * sign;
tempfrac = ((absraw >> 4) % 16) * 625;
minus = ((tempint == 0) & (sign == -1)) ? '-' : ' ';
//prilagojeni format izpisa
printf("- Senzor %c - %c%d.%04d\n", senzor_id, minus, tempint,
    tempfrac);
```

Ukaz *make serialdump* izpis shranjuje tudi v izbrano datoteko (kot smo opisali v podpoglavju 4.6) in na začetku doda čas v formatu Unix timestamp. Temu prilagodimo izpis, ki nam ga vrača element omrežja. Format zapisa v datoteki je tako:

....
1410639722 - Senzor 2 - 22.2500
1410639725 - Senzor 1 - 26.6875
1410639728 - Senzor 0 - 26.9375
1410639742 - Senzor 3 - 15.7500
1410639766 - Senzor 1 - 26.8125
1410639778 - Senzor 0 - 26.9375
1410639811 - Senzor 2 - 22.2500
....

V datoteki imamo sedaj vse podatke, ki jih potrebujemo. Rešitev, da to datoteko uporabimo za spremljanje omrežja z mobilnim telefonom je za

izdelavo prototipa najbolj smiselna.

Da lahko do datoteke dostopamo z mobilnim telefonom, mora biti dostopna preko interneta. To pomeni, da moramo postaviti nekaj zahtev za uspešno izdelavo prototipa: kjer imamo postavljeno BSO moramo imeti tudi računalnik z delujočo internetno povezavo. Na ta računalnik mora biti priklopljen eden od elementov omrežja. Tok podatkov in uporaba datoteke z izmerjenimi temperaturami sta prikazana na sliki 5.1.

Za dostopnost datoteke in njeno osveževanje v primeru spremembe (novega zapisa izmerjene temperature) smo uporabili storitev Dropbox[2], ki omogoča shranjevanje in deljenje datotek preko spleta ter sinhronizacijo in posodabljanje. Instalacija potrebne programske opreme v virtualnem stroju *Instant Contiki* ni zahtevna, vsa potrebna navodila lahko najdemo na spletni strani ponudnika storitve.

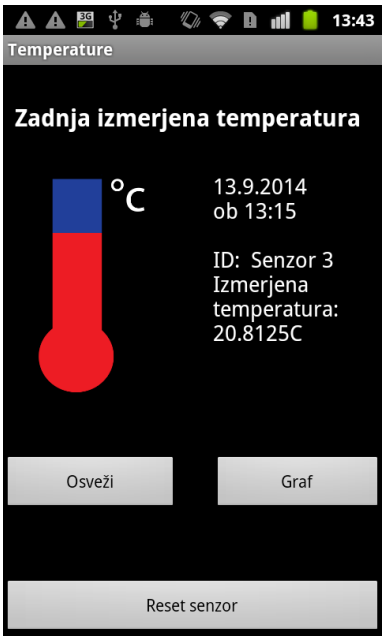
Naslednji korak pri izdelavi prototipa je izdelava mobilne aplikacije, ki nam bo omogočila spremljanje in nadzor nad vzpostavljenim BSO. Mobilna aplikacija mora omogočati jasen izpis zadnje izmerjene temperature in grafični prikaz izmerjenih temperatur za nek časovni interval. Kot dokaz koncepta nadzora omrežja z mobilnim telefonom pa mora omogočiti tudi povratni vpliv na BSO.

Izbrali smo izdelavo aplikacije na mobilnem telefonu z operacijskim sistemom Android, saj je med operacijskimi sistemi za pametne telefone njegov tržni delež največji. Za razvojno okolje smo uporabili Android SDK[3] in Eclipse, pametni telefon pa Nexus One.

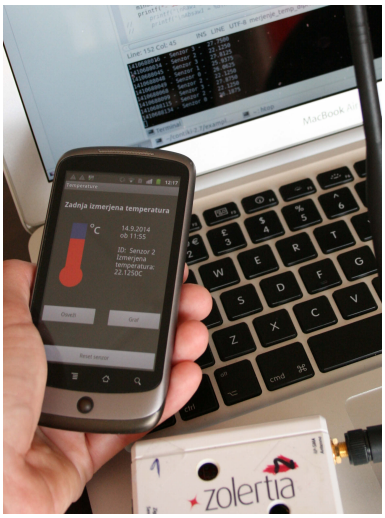
Z mobilno aplikacijo dostopamo do datoteke, jo preberemo ter ustrezno obdelamo podatke. Zadnja vrstica datoteke predstavlja zadnjo izmerjeno temperaturo in to tudi izpišemo v osnovni pogled aplikacije, kot je prikazano na slikah 5.3 in 5.4. S klikom oziroma dotikom na gumb *Osveži* ponovno preberemo datoteko in preverimo, ali je zapisana že kakšna nova vrstica. V kolikor je, izpišemo nov podatek.

[2]https://www.dropbox.com/

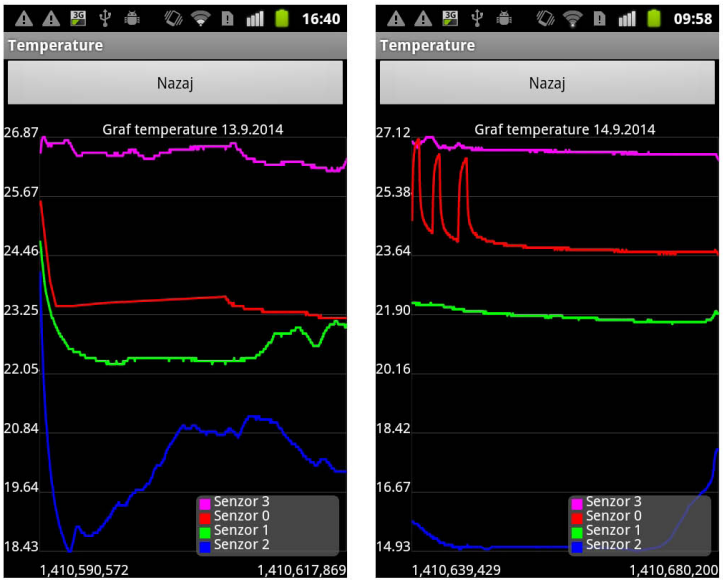
[3]http://developer.android.com/sdk/index.html



Slika 5.3: Osnovni pogled aplikacije



Slika 5.4: Pri začetnem testiranju pravilnosti in hitrosti delovanja izpisa zadnje izmerjene temperature v aplikaciji smo preverjali izpis v ukazni vrstici terminalskega okna z izpisom v mobilni aplikaciji.



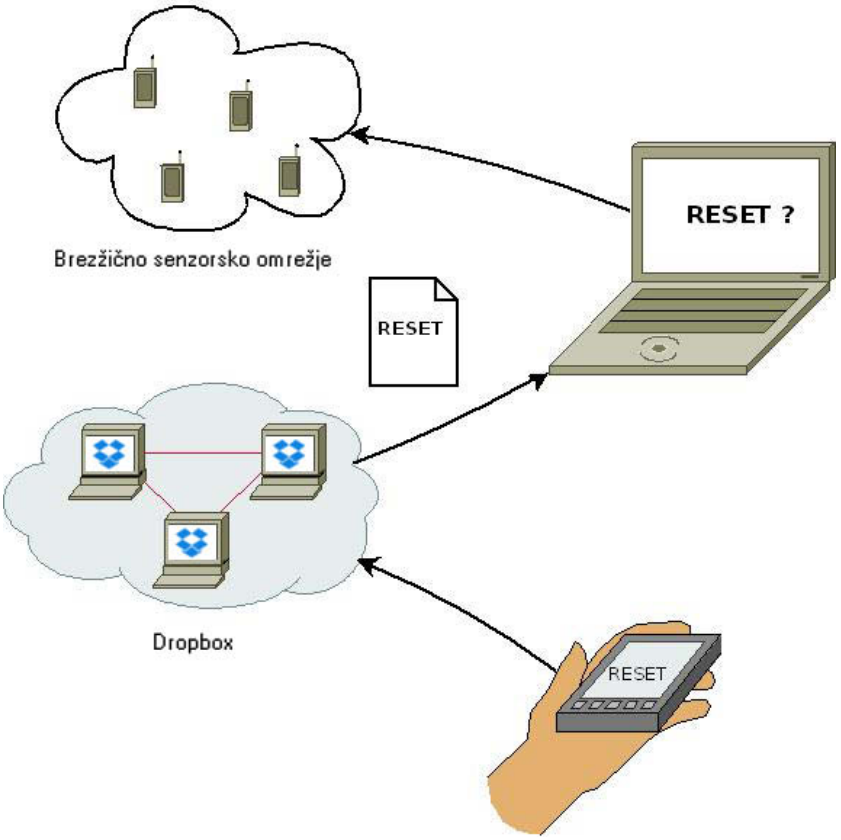
Slika 5.5: Graf izmerjene temperature

Z dotikom gumba z oznako *Graf* preklopimo na grafični prikaz (slika 5.5). Za izris grafičnega prikaza temperatur smo uporabili knjižnico GraphView[4].

Kot primer nadzora nad BSO smo si izbrali implementacijo ponovnega zagona elementa omrežja, ki je priključen na računalnik. Preko tega elementa spremljamo delovanje omrežja in lahko ustvarimo novo datoteko, kjer shranjujemo izmerjene temperature. Z meritvami tako z vidika uporabnika mobilne aplikacije začnemo ponovno. Za izvedbo tega dela smo uporabili storitve Dropbox API za Android. Izčrpna navodila za delo z API-jem so dostopna na *https://www.dropbox.com/developers/sync/start/android*.

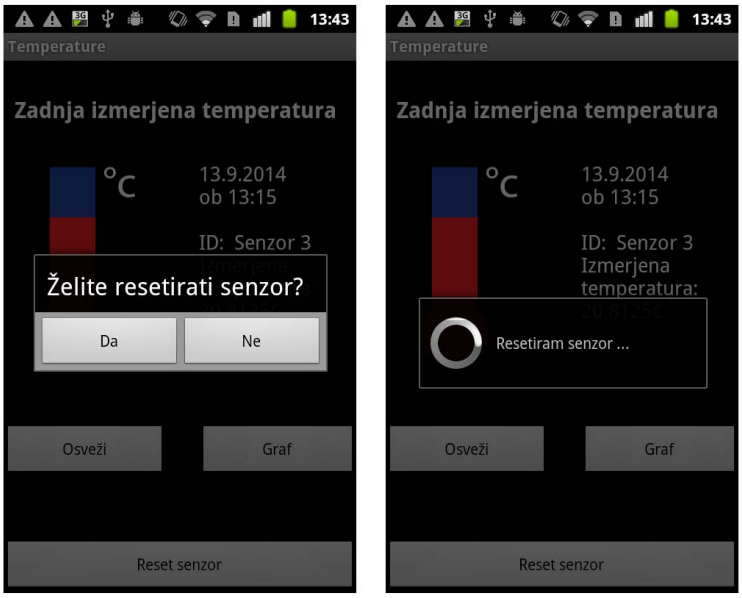
Osnovna zasnova koncepta je preprosta. Če želimo element omrežja resetirati, ustvarimo datoteko v določeni mapi in v primeru, da je datoteka prisotna tam, kjer to preverjamo, poženemo ukaz za reset, kot je prikazano na sliki 5.6. Datoteko nato po uspešnem resetu zbrišemo. S tem preprostim konceptom dosežemo, da mobilna naprava ozroma aplikacija ni vezana

[4]Knjižnica in navodila za uporabo so na voljo na spletni strani http://android-graphview.org/



Slika 5.6: Shema toka podatkov od aplikacije do ponovnega zagona elementa Z1 po pritisku na gumb *Reset senzor*

na določen računalnik, mora pa se povezati z določenim Dropbox računom. Spremljamo le spremembe v določni mapi, ki se sinhronizira z Dropbox strežnikom.



Slika 5.7: Resetiranje elementa omrežja s pomočjo mobilne aplikacije

Po pritisku na gumb *Reset senzor* se nam odpre okno z izbiro *Da* ali *Ne*, kot je prikazano na sliki 5.7. V primeru, da izberemo *Da*, aplikacija v ozadju ustvari novo datoteko *reset.txt* v izbrani mapi. Za to uporabimo DbxFileSystem, objekt ki ga uporabimo kot lokalni datotečni sistem, ki se sinhronizira z Dropbox strežnikom. Omogoča nam pregled datotek, ustvarjanje novih, premikanje, brisanje ipdr. Ko se procedura ponovnega zagona elementa omrežja izvrši, se vrnemo nazaj v glavno okno aplikacije.

Za uspešno izvedeno proceduro restart-a elementa potrebujemo še povezovalni člen med ustvarjeno datoteko in virtualnim strojem. To rešimo z bash skripto, s katero v virtualnem stroju *Instant Contiki* preverjamo obstoj te datoteke. V primeru, da datoteka obstaja, le-to zbrišemo in ponovno zaženemo element Zolertia Z1.

```
resetfile="/home/user/Dropbox/Apps/test_diplomska/reset.txt";
logfile="/home/user/Dropbox/Public/serialdump";
#klic ukaza serialdump izvedemo v skripti
make serialdump &
#v neskoncni zanki preverjamo obstoj datoteke na 5 sekund
#ali na drug poljuben interval
while : ;
do
  sleep 5
  #ali datoteka obstaja
  if [ -f  $resetfile ]
    then
      current_time=$(date "+%Y.%m.%d-%H.%M.%S-")
      #datoteko odstranimo
      rm $resetfile
      #ubijemo proces, ki ima odprto datoteko serialdump,
      #kjer logiramo izmerjeno temp. in ima v imenu serialdump
      kill -9 $(lsof -c serialdump -t $logfile)
      #za morebirno nadaljno uporabo log shranimo
      mv $logfile $current_time$logfile
      #reset
      cd /home/user/contiki-2.7/examples/ipv6/simple-udp-rpl/
      make z1-reset
      echo "restarting serialdump!"
      #ponoven klic ukaza
      make serialdump &
  fi
done
```

Naš prototip je mogoče zelo enostavno prenesti kamorkoli. Na računalnik namestimo virtualni stroj *Instant Contiki*, povežemo se z ustreznim računom Dropbox, prevedemo program za merjenje in pošiljanje temperature in zaženemo skripto. Na sliki 5.8 je prikazana vsa potrebna strojna oprema za izvedbo prototipa: element Zolertia Z1 (uporabili smo štiri elemente), povezana na računalnik z dostopom do interneta ter pametni telefon z Android OS.

Slika 5.8: Uporabljena strojna oprema za izvedbo prototipa BSO

Poglavje 6

Zaključek

V diplomski nalogi smo na splošno opisali lastnosti in pojasnili omejitve BSO. Ker je zaradi lastnosti teh omrežij nemogoče postaviti konkretno definicijo, ki bi ustrezala vsem takim omrežjem, sem pri opisovanju snovi naletela na veliko težav. Prepričana pa sem, da nam je uspelo sestaviti delo, ki bralcu sistematično in poljudno predstavi BSO in lahko služi kot podlaga za nadaljne delo.

Pri opisu strojne opreme smo se omejili na predstavitev elementov Zolertia Z1, ki so študentom na razpolago na Fakulteti za računalništvo in informatiko, UL. Pri izbiri strojne opreme za izgradno BSO imamo sicer zelo veliko možnosti. Lahko se odločimo za že končan element, kot je Zolertia Z1, ali pa sestavimo po svoji meri nov element. Zanimivo bi bilo primerjati različne elemente za BSO med seboj.

Predstavitev programske opreme smo zapisali predvsem z uporabniškega vidika, s čimer smo dosegli, da se lahko ta del diplomskega dela uporabi kot iztočnica za nadaljne delo s TinyOS ali Contiki operacijskim sistemom. Pri tem poglavju vidim prostor za izboljšavo predvsem v konkretnejši primerjavi, ne le z uporabniškega, temveč tudi razvojnega vidika. V nadaljevanju bi bilo tudi smiselno natančneje pregledati ostale možnosti pri izbiri operacijskega sistema za BSO. Nekatere izmed njih smo tudi na kratko opisali v v pričujočem delu. Še bolj zanimivo pa bi bilo izdelati nov koncept operacij-

skega sistema za izbrano strojno opremo.

V okviru praktičnega dela smo predstavili prototip BSO z uporabo strojne opreme Zolertia Z1 in operacijskega sistema Contiki. Povezali smo štiri elemente v BSO in ga nadzorovali s pomočjo mobilnega telefona z nameščenim Android OS. Ker gre za prototip, je možnosti za izboljšave zelo veliko. Če jih naštejemo le nekaj, pri sami identifikaciji senzorja bi lahko uporabili Node ID in spremembo MAC naslova v flash pomnilniku. Za komunikacijo bi lahko uporabili Rime protokolni sklad in optimalnejši način pošiljanja paketov. Na nivoju mobilne aplikacije je ena od enostavnejših izboljšav omogočiti izris grafa temperatur iz arhivskih datotek, torej za pretekle dni. Izpisovali bi lahko ne le zadnjo izmerjeno temperaturo, ampak zadnjo izmerjeno temperaturo vsakega senzorja.

Prepoznavnost brezžičnih senzorskih omrežj in njihova uporaba bo po mojem mnenju v prihodnjih letih strmo naraščala. Zanimivo bo videti, kako se bosta spreminjali strojna in programska oprema, uporabljeni v takih omrežjih. Ker gre za izredno široko področje predvidevam, da se bodo v prihodnosti pri načrtovanju različnih BSO prepletale številne znanosti – ne le računalništvo in informatika, temveč tudi ostale naravoslovne in morda tudi družboslovne vede.

Literatura

[1] H. Karl, A. Willig. *Protocols and architectures for wireless sensor networks*. John Wiley and sons, 2005.

[2] R. Verdone, D. Dardari, G. Mazzini, A. Conti. *Wireless sensor and actuator networks*. AP, 2008.

[3] S.K. Sarkar. *Wireless sensor and ad hoc networks under diversified network scenarios*. Artech house, 2012.

[4] P. Levis, D. Gay. *TinyOS Programming*. Cambridge University press, 2009.

[5] Wikipedia. Dostopno na:
http://en.wikipedia.org/

[6] J. Walters. *Wireless Sensor Network Security: A Survey*. Dostopno na:
http://www.eecis.udel.edu/ fei/reading/070426.wsn.security.survey.pdf

[7] Zolertia Wiki spletna stran. Dostopno na:
http://zolertia.sourceforge.net/wiki/index.php/Main_Page

[8] CC2420 Datasheet. Dostopno na:
http://www.ti.com/lit/ds/symlink/cc2420.pdf

[9] Zolertia Z1 Datasheet. Dostopno na:
http://zolertia.sourceforge.net/wiki/images/e/e8/Z1_RevC_Datasheet.pdf

[10] TMP102 Datasheet. Dostopno na:
http://www.ti.com/lit/ds/symlink/tmp102.pdf

[11] M. Farooq, T. Kunz. *Operating Systems for Wireless Sensor Networks: A Survey*. Dostopno na:
http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3231431/

[12] A. Dunkels, O. Schmidt, T. Voigt. *Using Protothreads for Sensor Node Programming*. Dostopno na:
http://dunkels.com/adam/dunkels05using.pdf

[13] TinyOS spletna stran. Dostopno na:
http://tinyos.stanford.edu/

[14] TOSThreads Tutorial. Dostopno na:
http://tinyos.stanford.edu/tinyos-wiki/index.php/TOSThreads_Tutorial

[15] How protothreads really work. Dostopno na:
http://dunkels.com/adam/pt/expansion.html

[16] T. Reusing. *Comparison of Operating Systems TinyOS and Contiki*. Dostopno na:
http://www.net.in.tum.de/fileadmin/TUM/NET/NET-2012-08-2/NET-2012-08-2_02.pdf

[17] A. K. Dwivedi, M. K. Tiwari, O. P. Vyas. *Operating Systems for Tiny Networked Sensors: A Survey*. Dostopno na:
http://academypublisher.com/ijrte/vol01/no02/ijrte0102152157.pdf

[18] W. Dargie. *Runtime Environments in Wireless Sensor Networks*. Dostopno na:
http://www.rn.inf.tu-dresden.de/lectures/WSN/runtime.pdf

[19] Contiki spletna stran. Dostopno na:
http://www.contiki-os.org/start.html

[20] Instant Contiki. Dostopno na:
http://sourceforge.net/projects/contiki/files/Instant%20Contiki/